∼ Topic VIII ∼

The state of the art
Scala

References:

- *Scala By Example* by M. Odersky. Programming Methods Laboratory, EPFL, 2008.
- *An overview of the Scala programming language* by M. Odersky *et al.* Technical Report LAMP-REPORT-2006-001, Second Edition, 2006.

Scala (I)

- Scala has been developed from 2001 in the Programming Methods Laboratory at EPFL by a group lead by Martin Odersky. It was first released publicly in 2004, with a second version released in 2006.
- Scala is aimed a the construction of components and component systems.

  One of the major design goals of Scala was that it should be flexible enough to act as a convenient host language for domain specific languages implemented by library modules.

- Scala has been designed to work well with Java and C#.

  Every Java class is seen in Scala as two entities, a class containing all dynamic members and a singleton object, containing all static members.

  Scala classes and objects can also inherit from Java classes and implement Java interfaces. This makes it possible to use Scala code in a Java framework.
- Scala's influences: Beta, C#, FamilyJ, gbeta, Haskell, Java, Jiazzi, $\text{ML}_{\leq}$, Moby, MultiJava, Nice, OCaml, Pizza, Sather, Smalltalk, SML, XQuery, *etc.*

A procedural language !

```
def qsort( xs: Array[Int] ) {
  def swap(i: Int, j:Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort(l: Int, r: Int) {
    val pivot = xs( (l+r)/2 ); var i = l; var j = r
    while (i <= j) {
      while ( lt( xs(i), pivot ) ) i += 1
      while ( lt( xs(j), pivot ) ) j -= 1
      if ( i<=j ) { swap(i,j); i += 1; j-= 1 }
    }
    if (l<j) sort(l,j)
    if (j<r) sort(i,r)
  }
  sort(0,xs.length-1)
}
```

NB:

- Definitions start with a reserved word.
- Type declarations use the colon notation.
- Array selections are written in functional notation.

  (In fact, arrays in Scala inherit from functions.)
- Block structure.

A declarative language !

```
def qsort[T]( xs: Array[T] )( lt: (T,T)=>Boolean ): Array[T]
= if ( xs.length <= 1 ) xs
  else {
    val pivot = xs( xs.length/2 )
    Array.concat( qsort( xs filter (x => lt(x,pivot)) ) lt ,
                         xs filter (x => x == pivot )      ,
                  qsort( xs filter (x => lt(pivot,x)) ) lt )
  }
```


NB:

- Polymorphism.
- Type declarations can often be omitted because the compiler can infer it from the context.
- Higher-order functions.
- The binary operation $e \star e'$ is always interpreted a the method call $e.\star(e')$.
- The equality operation == between values is designed to be transparent with respect to the type representation.

Scala (II)

Scala fuses (1) *object-oriented* and (2) *functional* programming in a statically typed programming language.

1. Scala uses a uniform and pure *object-oriented* model similar to that of Smalltalk: Every value is an object and every operation is a message send (that is, the invocation of a method).

   In fact, even primitive types are not treated specially; they are defined as type aliases of Scala classes.
2. Scala is also a *functional* language in the sense that functions are first-class values.

Mutable state

- Real-world objects with state are represented in Scala by objects that have variables as members.
- In Scala, all mutable state is ultimately built from variables.
- Every defined variable has to be initialised at the point of its definition.
- Variables may be `private`.

Blocks

Scala is an *expression-oriented* language, every function returns some result.

Blocks in Scala are themselves expressions. Every block ends in a result expression which defines its value.

Scala uses the usual block-structured scoping rules.

Functions

A function in Scala is a first-class value.

The anonymous function

```
( x1: T1, ... , xn: Tn ) => E
```

is equivalent to the block

```
{ def f ( x1: T1 , ... , xn: Tn ) = E ; f }
```

where `f` is a fresh name which is used nowhere else in the program.

Parameter passing

Scala uses call-by-value by default, but it switches to call-by-name evaluation if the parameter type is preceded by `=>`.

Imperative control structures

A functional implementation of while loops:

```
def whileLoop( cond: => Boolean )( comm: => Unit )
  { if (cond)  comm ; While( cond )( comm )  }
```

Classes and objects

- `class`es provide fields and methods. These are accessed using the dot notation. However, there may be `private` fields and methods that are inaccessible outside the class.

  Scala, being an object-oriented language, uses dynamic dispatch for method invocation. Dynamic method dispatch is analogous to higher-order function calls. In both cases, the identity of the code to be executed is known only at run-time. This similarity is not superficial. Indeed, Scala represents every function value as an object.

- Every class in Scala has a superclass which it `extends`.

  A class inherits all members from its superclass. It may also `override` (*i.e.* redefine) some inherited members.

  If class A extends class B, then objects of type A may be used wherever objects of type B are expected. We say in this case that type A *conforms* to type B.

- Scala maintains the invariant that interpreting a value of a subclass as an instance of its superclass does not change the representation of the value.

  Amongst other things, it guarantees that for each pair of types `S <: T` and each instance `s` of `S` the following equality holds:

  `s.asInstanceOf[T].asInstanceOf[S] = s`

- Methods in Scala do not necessarily take a parameter list. These *parameterless* methods are accessed just as value fields.

  The uniform access of fields and parameterless methods gives increased flexibility for the implementor of a class. Often, a field in one version of a class becomes a computed value in the next version. Uniform access ensures that clients do not have to be rewritten because of that change.

- `abstract class`es may have *deferred* members which are declared but which do not have an implementation. Therefore, no objects of an abstract class may be created using `new`.

```
abstract class IntSet {
  def incl( x:Int ): IntSet
  def contains( x:Int ): Boolean
}
```

Abstract classes may be used to provide interfaces.

♦ Scala has `object` definitions. An object definition defines a class with a single instance. It is not possible to create other objects with the same structure using `new`.

```
object EmptySet extends IntSet {
  def incl( x: Int ): IntSet
    = new NonEmptySet(x,EmptySet,EmptySet)
  def contains( x: Int ): Boolean = false
}
```

An object is created the first time one of its members is accessed. (This strategy is called *lazy evaluation*.)

♦ A `trait` is a special form of an abstract class that does not have any value parameters for its constructor and is meant to be combined with other classes.

```
trait IntSet {
  def incl( x:Int ): IntSet
  def contains( x:Int ): Boolean
}
```

Traits may be used to collect signatures of some functionality provided by different classes.

Case study (I)

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number( n: Int ) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}
```



```
class Sum( e1: Expr; e2: Expr ) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
def eval( e: Expr ): Int = {
  if (e.isNumber) e.NumValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("bad expression")
}
```

? What is good and what is bad about this implementation?

Case study (II)

```
abstract class Expr {
  def eval: Int
}
class Number( n: Int ) extends Expr {
  def eval: Int = n
}
class Sum( e1: Expr; e2: Expr ) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```


This implementation is easily extensible with *new types of data*:

```
class Prod( e1: Expr; e2: Expr ) extends Expr {
  def eval: Int = e1.eval * e2.eval
}
```

But, is this still the case for extensions involving *new operations* on existing data?

Case study (III)

Case classes

```
abstract class Expr
case class Number( n: Int ) extends Expr
case class Sum( e1: Expr; e2: Expr ) extends Expr
case class Prod( e1: Expr; e2: Expr ) extends Expr
```

- Case classes implicitly come with a constructor function, with the same name as the class.

  Hence one can construct expression trees as:

  ```
  Sum( Sum( Number(1) , Number(2) ) , Number(3) )
  ```


- Case classes and case objects implicitly come with implementations of methods `toString`, `equals`, and `hashCode`.
- Case classes implicitly come with nullary accessor methods which retrieve the constructor arguments.
- Case classes allow the constructions of *patterns* which refer to the case class constructor.

Case study (III)

Pattern matching

The `match` method takes as argument a number of cases:

```
def eval( e: Expr ): Int
  = e match
    {  case Number(x) => x
       case Sum(l,r) => eval(l) + eval(r)
       case Prod(l,r) => eval(l) * eval(r)
    }
```

If none of the patterns matches, the pattern matching expression is aborted with a `MatchError` exception.

Generic types and methods

- Classes in Scala can have type parameters.

```
abstract class Set[A] {
  def incl( x: A ): Set[A]
  def contains( x: A ): Boolean
}
```

- Scala has a fairly powerful type inferencer which allows one to omit type parameters to polymorphic functions and constructors.

Generic types

Variance annotations

The combination of type parameters and subtyping poses some interesting questions.

? If `T` is a subtype of a type `S`, should `Array[T]` be a subtype of the type `Array[S]`?

! No, if one wants to avoid run-time checks!

Example:

```
val x = new Array[String](1)
val y: Array[Any] = x  //  disallowed in Scala because
                       //  Array is not covariant
y.update( 0 , new Rational(1,2) )
```


In Scala, generic types like the following one:

```
class Array[A] {
  def apply( index: Int ): A
    ...
  def update( index: Int, elem: A )
    ...
}
```

have by default *non-variant* subtyping.

However, one can enforce *co-variant* subtyping by prefixing a formal type parameter with a `+`. There is also a prefix `-` which indicates *contra-variant* subtyping.

Scala uses a conservative approximation to verify soundness of variance annotations: a covariant type parameter of a class may only appear in co-variant position inside the class. Hence, the following class definition is rejected:

```
class Array[+A] {
  def apply( index: Int ): A
    ...
  def update( index:Int , elem: A )
    ...
}
```


Functions are objects

Recall that Scala is an object-oriented language in that every value is an object. It follows that *functions are objects* in Scala.

Indeed, the function type

$$(A_1 , \ldots , A_k) \Rightarrow B$$

is equivalent to the following parameterised class type:

```
abstract class Functionk[-A₁,...,-Aₖ,+B]
  { def apply( x₁:A₁,...,xₙ:Aₖ ): B }
```

> Since function types are classes in Scala, they can be further refined in subclasses. An example are arrays, which are treated as special functions over the type of integers.

The function `x => x+1` would be expanded to an instance of `Function1` as follows:

```
new Function1[Int,Int] {
  def apply( x:Int ): Int = x+1
}
```

Conversely, when a value of a function type is applied to some arguments, the `apply` method of the type is implicitly inserted; *e.g.* for `f` and object of type `Function1[A,B]`, the application `f(x)` is expanded to `f.apply(x)`.

NB: Function subtyping is contra-variant in its arguments whereas it is co-variant in its result. **?** Why?

Generic types

Type parameter bounds

```
trait Ord[A] {
  def lt( that: A ): Boolean
}
case class Num( value: Int ) extends Ord[Num] {
  def lt( that: Num ) = this.value < that.value
}
trait Heap[ A <: Ord[A] ] {
  def insert( x: A ): Heap[A]
  def min: A
  def remove: Heap[A]
}
```

Generic types
View bounds

One problem with type parameter bounds is that they require forethought: if we had not declared `Num` as a subclass of `Ord`, we would not have been able to use `Num` elements in heaps. By the same token, `Int` is not a subclass of `Ord`, and so integers cannot be used as heap elements.

A more flexible design, which admits elements of these types, uses *view bounds*:

```
trait Heap[ A <% Ord[A] ] {
  def insert( x: A ): Heap[A]
  def min: A
  def remove: Heap[A]
}
```

A view bounded type parameter clause `[ A <% T ]` only specifies that the bounded type `A` must be *convertible* to the bound type `T`, using an *implicit conversion*.

Views allow one to augment a class with new members and supported traits.

Generic types
Lower bounds

Co-variant generic functional stacks:

```
abstract class Stack[+A] {
  def push[B >: A]( x: B ): Stack[B]
    = new NonEmptyStack(x,this)
  def top: A
  def pop: Stack[A]
}
class NonEmptyStack[+A]( elem: A, rest: Stack[A] )
extends Stack[A] {
  def top = elem
  def pop = rest
}
```



```
object EmptyStack extends Stack[Nothing] {
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
```

- Scala does not allow to parameterise objects with types.
- `Nothing` is a subtype of all other types.

Implicit parameters and conversions

- **Implicit parameters**

  In Scala, there is an `implicit` keyword that can be used at the beginning of a parameter list.

```
def qsort[T]( xs: Array[T] )( implicit o: Ord[T] ): Array[T]
= if ( xs.length <= 1 ) xs
  else {
    val pivot = xs( xs.length/2 )
    Array.concat( qsort( xs filter (x => o.lt(x,pivot)) ) ,
                         xs filter (x => x == pivot )     ,
                  qsort( xs filter (x => o.lt(pivot,x)) ) )
  }
```


The principal idea behind implicit parameters is that arguments for them can be left out from a method call. If the arguments corresponding to implicit parameters are missing, they are inferred by the Scala compiler.

NB: View bounds are convenient syntactic sugar for implicit parameters.

- **Implicit conversions**

  As last resort in case of type mismatch the Scala compiler will try to apply an implicit conversion.

```
implicit def int2ord( x: Int ): Ord[Int]
  = new Ord[Int] { def lt( y: Int ) = x < y }
```

Implicit conversions can also be applied in member selections.

Mixin-class composition

Every class or object in Scala can inherit from several traits in addition to a normal class.

```
trait AbsIterator[T] {
  def hasNext: Boolean
  def next: T
}
trait RichIterator[T] extends AbsIterator[T] {
  def foreach( f: T=>Unit ): Unit =
    while (hasNext) f(next)
}
```



```
class StringIterator( s: String )
  extends AbsIterator[Char] {
    private var i = 0
    def hasNext = i < s.length
    def next = { val x = s charAt i; i = i+1; x }
  }
```

Traits can be used in all contexts where other abstract classes appear; however only traits can be used as *mixins*.

```
object Test {
  def main( args: Array[String] ): Unit = {
    class Iter extends StringIterator(args(0))
               with    RichIterator[Char]
    val iter = new Iter
    iter foreach System.out.println
  }
}
```

The class `Iter` is constructed from a *mixin composition* of the parents `StringIterator` (called the *superclass*) and `RichIterator` (called a *mixin*) so as to combine their functionality.

The class `Iter` inherits members from both `StringIterator` and `RichIterator`.

NB: Mixin-class composition is a form of multiple inheritance!

Language innovations

- Flexible syntax and type system.
- Pattern matching over class hierarchies unifies functional and object-oriented data access.
- Abstract types and mixin composition unify concepts from object and module systems.