A Tour of the Scala Programming Language

Version 1.6
January 22, 2007

Martin Odersky
Vincent Cremet
Iulian Dragos
Gilles Dubochet
Burak Emir
Philipp Haller
Sean McDirmid
Stéphane Micheloud
Nikolay Mihaylov
Lex Spoon
Matthias Zenger

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

Contents

The Scala Programming Language

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.

Scala is object-oriented

Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits. Class abstractions are extended by subclassing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

Scala is functional

Scala is also a functional language in the sense that every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying. Scala's case classes and its built-in support for pattern matching model algebraic types used in many functional programming languages.
Furthermore, Scala's notion of pattern matching naturally extends to the processing of XML data with the help of right-ignoring sequence patterns. In this context, sequence comprehensions are useful for formulating queries. These features make Scala ideal for developing applications like web services (external link).

Scala is statically typed

Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner. In particular, the type system supports:

- generic classes,
- variance annotations,
- upper and lower type bounds,
- inner classes and abstract types as object members,
- compound types,
- explicitly typed self references,
- views, and
- polymorphic methods.

A local type inference mechanism takes care that the user is not required to annotate the program with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

Scala is extensible

The design of Scala acknowledges the fact that in practice, the development of domain specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries:

- any method may be used as an infix or postfix operator, and
- closures are constructed automatically depending on the expected type (target typing).

A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

Scala interoperates with Java and .NET

Scala is designed to interoperate well with popular programming environments like the Java 2 Runtime Environment (JRE) and the .NETFramework (CLR). In particular, the interaction with mainstream object-oriented languages like Java and C#is as smooth as possible. Scala has the same compilation model (separate compilation, dynamic class loading) like Java and C#and allows access to thousands of high-quality libraries.

Abstract Types

In Scala, classes are *parameterized* with values (the constructor parameters) and with types (if classes are generic). For reasons of regularity, it is not only possible to have values as *object members*; types along with values are members of objects. Furthermore, both forms of members can be concrete and abstract.
Here is an example which defines both a deferred value definition and an *abstract type* definition as members of class `Buffer`.

```
abstract class Buffer {
  type T
  val element: T
}
```

Abstract types are types whose identity is not precisely known. In the example above, we only know that each object of class `Buffer` has a type member `T`, but the definition of class `Buffer` does not reveal to what concrete type the member type `T` corresponds. Like value definitions, we can override type definitions in subclasses. This allows us to reveal more information about an abstract type by tightening the type bound (which describes possible concrete instantiations of the abstract type). In the following program we derive a class `SeqBuffer` which allows us to store only sequences in the buffer by stating that type `T` has to be a subtype of a new abstract type `U`:

```
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

Traits or classes with abstract type members are often used in combination with anonymous class instantiations. To illustrate this, we now look at a program which deals with a sequence buffer that refers to a list of integers:

```
abstract class IntSeqBuffer extends SeqBuffer {
  type U = Int
}
object AbstractTypeTest1 extends Application {
  def newIntSeqBuf(elem1: Int, elem2: Int): IntSeqBuffer =
    new IntSeqBuffer {
      type T = List[U]
      val element = List(elem1, elem2)
    }
  val buf = newIntSeqBuf(7, 8)
  Console.println("length = " + buf.length)
  Console.println("content = " + buf.element)
}
```

The return type of method `newIntSeqBuf` refers to a specialization of trait `Buffer` in which type `U` is now equivalent to `Int`. We have a similar *type alias* in the anonymous class instantiation within the body of method `newIntSeqBuf`. Here we create a new instance of `IntSeqBuffer` in which type `T` refers to `List[Int]`.
Please note that it is often possible to turn abstract type members into type parameters of classes and vice versa. Here is a version of the code above which only uses type parameters:

```
abstract class Buffer[+T] {
  val element: T
}
abstract class SeqBuffer[U, +T <: Seq[U]] extends Buffer[T] {
  def length = element.length
}
object AbstractTypeTest2 extends Application {
  def newIntSeqBuf(e1: Int, e2: Int): SeqBuffer[Int, Seq[Int]] =
    new SeqBuffer[Int, List[Int]] {
      val element = List(e1, e2)
    }
  val buf = newIntSeqBuf(7, 8)
  Console.println("length = " + buf.length)
  Console.println("content = " + buf.element)
}
```

Note that we have to use variance annotations here; otherwise we would not be able to hide the concrete sequence implementation type of the object returned from method `newIntSeqBuf`. Furthermore, there are cases where it is not possible to replace abstract types with type parameters.

Attribute Clauses

Attributes associate meta-information with definitions.
A simple attribute clause has the form `[C]` or `[C(a1, .., an)]`. Here, `C` is a constructor of a class `C`, which must conform to the class `scala.Attribute`. All given constructor arguments `a1, .., an` must be constant expressions (i.e., expressions on numeral literals, strings, class literals, Java enumerations and one-dimensional arrays of them).
An attribute clause applies to the first definition or declaration following it. More than one attribute clause may precede a definition and declaration. The order in which these clauses are given does not matter.
The meaning of attribute clauses is *implementation-dependent*. On the Java platform, the following Scala attributes have a standard meaning.

Scala	Java
scala.reflect.BeanProperty	Design pattern
scala.serializable	java.io.Serializable
scala.cloneable	java.lang.Cloneable
scala.deprecated (since 2.3.2)	java.lang.Deprecated
scala.remote	java.rmi.Remote
scala.transient	transient (keyword)
scala.volatile	volatile (keyword)
scala.SerialVersionUID	serialVersionUID (field)
scala.throws	throws (keyword)

In the following example we add the `throws` attribute to the definition of the method `read` in order to catch the thrown exception in the Java main program.

```
package examples
import java.io._
class Reader(fname: String) {
  private val in = new BufferedReader(new FileReader(fname))
  [throws(classOf[IOException])]
  def read() = in.read()
}
```

The following Java program prints out the contents of the file whose name is passed as the first argument to the `main` method.

```
package test;
import examples.Reader;  // Scala class !!
public class AttribTest {
    public static void main(String[] args) {
        try {
            Reader in = new Reader(args[0]);
            int c;
```

```
            while ((c = in.read()) != -1) {
                System.out.print((char) c);
            }
        } catch (java.io.Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Commenting out the `throws` attribute in the class `Reader` produces the following error message when compiling the Java main program:

```
Main.java:11: exception java.io.IOException is never thrown in body of
corresponding try statement
        } catch (java.io.IOException e) {
          ^
1 error
```

Java annotations

Note: Make sure you use the `-target:jvm-1.5` option with Java annotations.
Java 1.5 introduced user-defined metadata in the form of *annotations*. A key feature of annotations is that they rely on specifying name-value pairs to initialize their elements. For instance, if we need an annotation to track the source of some class we might define it as

```
@interface Source {
  public String URL();
  public String mail();
}
```

And then apply it as follows

```
@Source( URL = "http://coders.com/" ,
         mail = "support@coders.com")
public class MyClass extends HisClass ...
```

An attribute application in Scala looks like a constructor invocation. This is not enough for Java annotations so the syntax has been extended in a way pretty much similar to Java's

```
[Source { val URL = "http://coders.com/" ,
          val mail = "support@coders.com" }]
class MyScalaClass ...
```

This syntax is quite tedious if the annotation contains only one element (without default value) so, by convention, if the name is specified as `value` it can be applied in Java using a constructor-like syntax:

```
@interface SourceURL {
    public String value();
    public String mail() default "";
}
```

And then apply it as follows

```
@SourceURL("http://coders.com/")
public class MyClass extends HisClass ...
```

In this case, Scala provides the same possibility

```
[SourceURL("http://coders.com/")]
class MyScalaClass ...
```

The `mail` element was specified with a default value so we need not explicitly provide a value for it. However, if we need to do it we can not mix-and-match the two styles in Java:

```
@SourceURL( value = "http://coders.com/" ,
            mail = "support@coders.com")
public class MyClass extends HisClass ...
```

Scala provides more flexibility in this respect

```
[SourceURL("http://coders.com/")
          { val mail = "support@coders.com" }]
class MyScalaClass ...
```

This extended syntax is consistent with .NET's attributes and can accomodate their full capabilites.

Case Classes

Scala supports the notion of *case classes*. Case classes are regular classes which export their constructor parameters and which provide a recursive decomposition mechanism via pattern matching.
Here is an example for a class hierarchy which consists of an abstract super class `Term` and three concrete case classes `Var`, `Fun`, and `App`.

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

This class hierarchy can be used to represent terms of the untyped lambda calculus. To facilitate the construction of case class instances, Scala does not require that the **new** primitive is used. One can simply use the class name as a function.
Here is an example:

```
Fun("x", Fun("y", App(Var("x"), Var("y"))))
```

The constructor parameters of case classes are treated as public values and can be accessed directly.

```
val x = Var("x")
Console.println(x.name)
```

For every case class the Scala compiler generates `equals` method which implements structural equality and a `toString` method. For instance:

```
val x1 = Var("x")
val x2 = Var("x")
val y1 = Var("y")
Console.println("" + x1 + " == " + x2 + " => " + (x1 == x2))
Console.println("" + x1 + " == " + y1 + " => " + (x1 == y1))
```

will print

```
Var(x) == Var(x) => true
Var(x) == Var(y) => false
```

It makes only sense to define case classes if pattern matching is used to decompose data structures. The following object defines a pretty printer function for our lambda calculus representation:

```
object TermTest extends Application {
  def print(term: Term): Unit = term match {
    case Var(n) =>
      Console.print(n)
    case Fun(x, b) =>
      Console.print("^" + x + ".")
      print(b)
    case App(f, v) =>
      Console.print("(")
      print(f)
      Console.print(" ")
      print(v)
      Console.print(")")
  }
  def isIdentityFun(term: Term): Boolean = term match {
    case Fun(x, Var(y)) if x == y => true
    case _ => false
  }
  val id = Fun("x", Var("x"))
  val t = Fun("x", Fun("y", App(Var("x"), Var("y"))))
  print(t)
  Console.println
  Console.println(isIdentityFun(id))
  Console.println(isIdentityFun(t))
}
```

In our example, the function `print` is expressed as a pattern matching statement starting with the **`match`** keyword and consisting of sequences of **`case`** `Pattern =>` `Body` clauses.
The program above also defines a function `isIdentityFun` which checks if a given term corresponds to a simple identity function. This example uses deep patterns and guards. After matching a pattern with a given value, the guard (defined after the keyword **`if`**) is evaluated. If it returns `true`, the match succeeds; otherwise, it fails and the next pattern will be tried.

Classes

Classes in Scala are static templates that can be instantiated into many objects at runtime.
Here is a class definition which defines a class `Point`:

```
class Point(xc: Int, yc: Int) {
  var x: Int = xc
  var y: Int = yc
  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }
  override def toString(): String = "(" + x + ", " + y + ")";
}
```

The class defines two variables `x` and `y` and two methods: `move` and `toString`. `move` takes two integer arguments but does not return a value (the return type `Unit` corresponds to `void` in Java-like languages). `toString`, on the other hand, does not take any parameters but returns a `String` value. Since `toString` overrides the predefined `toString` method, it has to be tagged with the `override` flag.
Classes in Scala are parameterized with constructor arguments. The code above defines two constructor arguments, `xc` and `yc`; they are both visible in the whole body of the class. In our example they are used to initialize the variables `x` and `y`.
Classes are instantiated with the **`new`** primitive, as the following example will show:

```
object Classes {
  def main(args: Array[String]): Unit = {
    val pt = new Point(1, 2)
    Console.println(pt)
    pt.move(10, 10)
    Console.println(pt)
  }
}
```

The program defines an executable application `Classes` in form of a top-level singleton object with a `main` method. The `main` method creates a new `Point` and stores it in value `pt`. Note that values defined with the **`val`** construct are different from variables defined with the **`var`** construct (see class `Point` above) in that they do not allow updates; i.e. the value is constant.
Here is the output of the program:

```
(1, 2)
(11, 12)
```

Predefined function classOf

The predefined function classOf[T] returns a runtime representation of the Scala class type T.
The following Scala code example prints out the runtime representation of the args parameter:

```
object ClassReprTest {
  abstract class Bar {
    type T <: AnyRef
    def bar(x: T): Unit = {
      Console.println("5: " + x.getClass())
    }
  }
  def main(args: Array[String]): Unit = {
    Console.println("1: " + args.getClass())
    Console.println("2: " + classOf[Array[String]])
    new Bar {
      type T = Array[String]
      val x: T = args
      Console.println("3: " + x.getClass())
      Console.println("4: " + classOf[T])
    }.bar(args)
  }
}
```

Here is the output of the Scala program:

```
1: class [Ljava.lang.String;
2: class [Ljava.lang.String;
3: class [Ljava.lang.String;
4: class [Ljava.lang.String;
5: class [Ljava.lang.String;
```

A similar code example in Java 5 (aka. JDK 1.5) looks as follows:
When you execute the above Java program it will display four times the string "class [Ljava.lang.String;".

Compound Types

Sometimes it is necessary to express that the type of an object is a subtype of several other types. In Scala this can be expressed with the help of *compound types*, which are intersections of object types.
Suppose we have two traits `Cloneable` and `Resetable`:

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = { super.clone(); this }
}
trait Resetable {
  def reset: Unit
}
```

Now suppose we want to write a function `cloneAndReset` which takes an object, clones it and resets the original object:

```
def cloneAndReset(obj: ?): Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

The question arises what the type of the parameter `obj` is. If it's `Cloneable` then the object can be cloned, but not reset; if it's `Resetable` we can reset it, but there is no clone operation. To avoid type casts in such a situation, we can specify the type of `obj` to be both `Cloneable` and `Resetable`. This compound type is written like this in Scala: `Cloneable with Resetable`.
Here's the updated function:

```
def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  //...
}
```

Compound types can consist of several object types and they may have a single *refinement* which can be used to narrow the signature of existing object members.
The general form is: *A* `with` *B* `with` *C* `...` *refinement*
An example for the use of refinements is given on the page about abstract types.

Sequence Comprehensions

Scala offers a lightweight notation for expressing sequence comprehensions. Comprehensions have the form `for` (*enums*) `yield` *e*, where *enums* refers to a semicolon-separated list of enumerators. An enumerator is either a generator which introduces new variables, or it is a filter. A comprehension evaluates the body *e* for each binding generated by the enumerators *enum* and returns a sequence of these values. Here is an example:

```
object ComprehensionTest1 extends Application {
  def even(from: Int, to: Int): List[Int] =
    for (val i <- List.range(from, to); i % 2 == 0) yield i
  Console.println(even(0, 20))
}
```

The for-expression in function even introduces a new variable `i` of type `Int` which is subsequently bound to all values of the list `List`(*from*, *from* + 1, ..., *to* - 1). The constraint `i % 2 == 0` filters out all odd numbers so that the body (which only consists of the expression *i*) is only evaluated for even numbers. Consequently, the whole for-expression returns a list of even numbers.
The program yields the following output:

```
List(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

Here is a more complicated example which computes all pairs of numbers between 0 and *n*-1 whose sum is equal to a given value *v*:

```
object ComprehensionTest2 extends Application {
  def foo(n: Int, v: Int): Iterator[Pair[Int, Int]] =
    for (val i <- Iterator.range(0, n);
         val j <- Iterator.range(i + 1, n);
         i + j == v) yield
      Pair(i, j);
  foo(20, 32) foreach {
    case Pair(i, j) =>
      Console.println("(" + i + ", " + j + ")")
  }
}
```

This example shows that comprehensions are not restricted to lists. The previous program uses iterators instead. Every datatype that supports the operations `filter`, `map`, and `flatMap` (with the proper types) can be used in sequence comprehensions. Here's the output of the program:

```
(13, 19)
(14, 18)
(15, 17)
```

There is also a special form of sequence comprehension which returns `Unit`. Here the bindings that are created from the list of generators and filters are used to perform side-effects. The programmer has to omit the keyword **`yield`** to make use of such a sequence comprehension.
Here's a program which is equivalent to the previous one but uses the special for comprehension returning `Unit`:

```
object ComprehensionTest3 extends Application {
  for (val i <- Iterator.range(0, 20);
       val j <- Iterator.range(i + 1, 20);
       i + j == 32)
    Console.println("(" + i + ", " + j + ")")
}
```

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.
Here is an example:

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def modN(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  Console.println(filter(nums, modN(2)))
  Console.println(filter(nums, modN(3)))
}
```

Note that method `modN` is partially applied in the two `filter` calls; i.e. only its first argument is actually applied. The term `modN(2)` yields a function of type `Int => Boolean` and is thus a possible candidate for the second argument of function `filter`.
Here's the output of the program above:

```
List(2,4,6,8)
List(3,6)
```

Extractor Objects

Scala allows the definition of patterns independently of case classes, using `unapply` methods in extractor objects.
Here is an example:

```
object Twice {
  def apply(x: Int) = x * 2
  def unapply(z: Int) = if (z%2 == 0) Some(z/2) else None
}

object TwiceTest extends Application {
  val x = Twice(21)
  x match { case Twice(n) => Console.println(n) } // prints 21
}
```

In the above example, `Twice` is an extractor object with two methods:

- The `apply` method is used to build even numbers.
- The `unapply` method is used to decompose an even number; it is in a sense the reverse of `apply`. `unapply` methods return option types: `Some(...)` for a match that suceeds, `None` for a match that fails. Pattern variables are returned as the elements of `Some`. If there are several variables, they are grouped in a tuple.

In the second-to-last line, `Twice`'s `apply` method is used to construct a number `x`. In the last line, `x` is tested against the pattern `Twice(n)`. This pattern succeeds for even numbers and assigns to the variable `n` one half of the number that was tested. The pattern match makes use of the `unapply` method of object `Twice`.
More details on extractors can be found in the paper "Matching Objects with Patterns" by Emir, Odersky and Williams (January 2007).

Nested Functions

In Scala it is possible to nest function definitions. The following object provides a filter function for extracting values from a list of integers that are below a threshold value:

```
object FilterTest extends Application {
  def filter(xs: List[Int], threshold: Int) = {
    def process(ys: List[Int]): List[Int] =
      if (ys.isEmpty) ys
      else if (ys.head < threshold) ys.head :: process(ys.tail)
      else process(ys.tail)
    process(xs)
  }
  Console.println(filter(List(1, 9, 2, 8, 3, 7, 4), 5))
}
```

Note that the nested function `process` refers to variable `threshold` defined in the outer scope as a parameter value of `filter`.
The output of this program is:

```
List(1,2,3,4)
```

Anonymous Function Syntax

Scala provides a relatively lightweight syntax for defining anonymous functions. The following expression creates a successor function for integers:

```
(x: Int) => x + 1
```

This is a shorthand for the following anonymous class definition:

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

It is also possible to define functions with multiple parameters:

```
(x: Int, y: Int) => "(" + x + ", " + y + ")"
```

or with no parameter:

```
() => { System.getProperty("user.dir") }
```

There is also a very lightweight way to write function types. Here are the types of the three functions defined above:

```
Int => Int
(Int, Int) => String
Unit => String
```

This syntax is a shorthand for the following types:

```
Function1[Int, Int]
Function2[Int, Int, String]
Function1[Unit, String]
```

Generic Classes

Like in Java 5 (aka. JDK 1.5), Scala has built-in support for classes parameterized with types. Such generic classes are particularly useful for the development of collection classes.
Here is an example which demonstrates this:

```
class Stack[T] {
  var elems: List[T] = Nil
  def push(x: T): Unit = elems = x :: elems
  def top: T = elems.head
  def pop: Unit = elems = elems.tail
}
```

Class `Stack` models imperative (mutable) stacks of an arbitrary element type `T`. The use of type parameters allows to check that only legal elements (that are of type `T`) are pushed onto the stack. Similarly, with type parameters we can express that method `top` will only yield elements of the given type.
Here are some usage examples:

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

The output of this program will be:

```
97
1
```

Note that subtyping of generic types is *invariant*. This means that if we have a stack of characters of type `Stack[Char]` then it cannot be used as an integer stack of type `Stack[Int]`. This would be unsound because it would enable us to enter true integers into the character stack. To conclude, *Stack[T]* is only a subtype of *Stack[S]* iff $S = T$. Since this can be quite restrictive, Scala offers a type parameter annotation mechanism to control the subtyping behavior of generic types.

Higher-Order Functions

Scala allows the definition of higher-order functions. These are functions that take other functions as parameters, or whose result is a function. Here is a function `apply` which takes another function `f` and a value `v` and applies function `f` to `v`:

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

Note that methods are automatically coerced to functions if the context requires this.
Here is an example:

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

object FunTest extends Application {
  def apply(f: Int => String, v: Int) = f(v)
  val decorator = new Decorator("[", "]")
  Console.println(apply(decorator.layout, 7))
}
```

Execution yields the output:

```
[7]
```

In this example, the method `decorator.layout` is coerced automatically to a value of type `Int => String` as required by method `app`. Please note that method `decorator` 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.

Implicit Parameters

A method with implicit parameters can be applied to arguments just like a normal method. In this case the **implicit** label has no effect. However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided.
The actual arguments that are eligible to be passed to an implicit parameter fall into two categories:

- First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition or an implicit parameter.
- Second, eligible are also all members of companion modules of the implicit parameter's type that are labeled **implicit**.

In the following example we define a method `sum` which computes the sum of a list of elements using the monoid's `add` and `unit` operations. Please note that implicit values can not be top-level, they have to be members of a template.

```
abstract class SemiGroup[A] {
  def add(x: A, y: A): A
}
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
}
object ImplicitTest extends Application {
  implicit object StringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object IntMonoid extends Monoid[int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))

  sum(List(1, 2, 3))
  sum(List("a", "b", "c"))
}
```

Local Type Inference

Scala has a built-in type inference mechanism which allows the programmer to omit certain type annotations. It is, for instance, often not necessary in Scala to specify the type of a variable, since the compiler can deduce the type from the initialization expression of the variable. Also return types of methods can often be omitted since they corresponds to the type of the body, which gets inferred by the compiler.
Here is an example:

```
object InferenceTest1 extends Application {
  val x = 1 + 2 * 3         // the type of x is Int
  val y = x.toString()      // the type of y is String
  def succ(x: Int) = x + 1  // method succ returns Int values
}
```

For recursive methods, the compiler is not able to infer a result type. Here is a program which will fail the compiler for this reason:

```
object InferenceTest2 {
  def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)
}
```

It is also not compulsory to specify type parameters when polymorphic methods are called or generic classes are instantiated. The Scala compiler will infer such missing type parameters from the context and from the types of the actual method/constructor parameters.
Here is an example which illustrates this:

```
case class MyPair[A, B](x: A, y: B);
object InferenceTest3 extends Application {
  def id[T](x: T) = x
  val p = new MyPair(1, "scala") // type: MyPair[Int, String]
  val q = id(1)                  // type: Int
}
```

The last two lines of this program are equivalent to the following code where all inferred types are made explicit:

```
  val x: MyPair[Int, String] = new MyPair[Int, String](1, "scala")
  val y: Int = id[Int](1)
```

In some situations it can be quite dangerous to rely on Scala's type inference mechanism as the following program shows:

```
object InferenceTest4 {
  var obj = null
  obj = new Object()
}
```

This program does not compile because the type inferred for variable `obj` is `AllRef`. Since the only value of that type is **`null`**, it is impossible to make this variable refer to another value.

Inner Classes

In Scala it is possible to let classes have other classes as members. Opposed to Java-like languages where such inner classes are members of the *enclosing class*, in Scala such inner classes are bound to the *outer object*. To illustrate the difference, we quickly sketch the implementation of a graph datatype:

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node): Unit =
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes;
      }
  }
  var nodes: List[Node] = Nil;
  def newNode: Node = {
    var res = new Node
    nodes = res :: nodes
    res
  }
}
```

In our program, graphs are represented by a list of nodes. Nodes are objects of inner class `Node`. Each node has a list of neighbours, which get stored in the list `connectedNodes`. Now we can set up a graph with some nodes and connect the nodes incrementally:

```
object GraphTest extends Application {
  val g = new Graph
  val n1 = g.newNode
  val n2 = g.newNode
  val n3 = g.newNode
  n1.connectTo(n2)
  n3.connectTo(n1)
}
```

We now enrich the following example with types to state explicitly what the type of the various defined entities is:

```
object GraphTest extends Application {
  val g: Graph = new Graph
  val n1: g.Node = g.newNode
  val n2: g.Node = g.newNode
  val n3: g.Node = g.newNode
  n1.connectTo(n2)
  n3.connectTo(n1)
}
```

This code clearly shows that a node type is prefixed with its outer instance (which is object g in our example). If we now have two graphs, the type system of Scala does not allow us to mix nodes defined within one graph with the nodes of another graph, since the nodes of the other graph have a different type.
Here is an illegal program:

```
object IllegalGraphTest extends Application {
  val g: Graph = new Graph
  val n1: g.Node = g.newNode
  val n2: g.Node = g.newNode
  n1.connectTo(n2)      // legal
  val h: Graph = new Graph
  val n3: h.Node = h.newNode
  n1.connectTo(n3)      // illegal!
}
```

Please note that in Java the last line in the previous example program would have been correct. For nodes of both graphs, Java would assign the same type `Graph.Node`; i.e. `Node` is prefixed with *class* `Graph`. In Scala such a type can be expressed as well, it is written `Graph#Node`. If we want to be able to connect nodes of different graphs, we have to change the definition of our initial graph implementation in the following way:

```
class Graph {
  class Node {
    var connectedNodes: List[Graph#Node] = Nil;
    def connectTo(node: Graph#Node): Unit =
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
  }
  var nodes: List[Node] = Nil;
  def newNode: Node = {
    var res = new Node
    nodes = res :: nodes
    res
  }
}
```

Please note that this program doesn't allow us to attach a node to two different graphs. If we want to remove this restriction as well, we have to change the type of variable `nodes` and the return type of method `newNode` to `Graph#Node`.

Lower Type Bounds

While upper type bounds limit a type to a subtype of another type, lower type bounds declare a type to be a supertype of another type. The term `T >: A` expresses that the type parameter `T` or the abstract type `T` refer to a supertype of type `A`.
Here is an example where this is useful:

```
case class ListNode[T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend(elem: T): ListNode[T] =
    ListNode(elem, this)
}
```

The program above implements a linked list with a `prepend` operation. Unfortunately, this type is invariant in the type parameter of class `ListNode`; i.e. type `ListNode` is *not* a subtype of type `List[Object]`. With the help of variance annotations we can express such a subtype semantics:

```
case class ListNode[+T](h: T, t: ListNode[T]) { ... }
```

Unfortunately, this program does not compile, because a *covariance* annotation is only possible if the type variable is used only in covariant positions. Since type variable `T` appears as a parameter type of method `prepend`, this rule is broken. With the help of a lower type bound, though, we can implement a `prepend` method where `T` only appears in covariant positions.
Here is the corresponding code:

```
case class ListNode[+T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend[U >: T](elem: U): ListNode[U] =
    ListNode(elem, this)
}
```

Note that the new `prepend` method has a slightly less restrictive type. It allows, for instance, to prepend an object of a supertype to an existing list. The resulting list will be a list of this supertype.
Here is some code which illustrates this:

```
object LowerBoundTest extends Application {
  val empty: ListNode[AllRef] = ListNode(null, null)
  val strList: ListNode[String] = empty.prepend("hello")
                                       .prepend("world")
  val anyList: ListNode[Any] = strList.prepend(12345)
}
```

Mixin Class Composition

As opposed to languages that only support *single inheritance*, Scala has a more general notion of class reuse. Scala makes it possible to reuse the *new member definitions of a class* (i.e. the delta in relationship to the superclass) in the definition of a new class. This is expressed as a *mixin-class composition*. Consider the following abstraction for iterators.

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
}
```

Next, consider a mixin class which extends `AbsIterator` with a method `foreach` which applies a given function to every element returned by the iterator. To define a class that can be used as a mixin we use the keyword **`trait`**.

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

Here is a concrete iterator class, which returns successive characters of a given string:

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

We would like to combine the functionality of `StringIterator` and `RichIterator` into a single class. With single inheritance and interfaces alone this is impossible, as both classes contain member impementations with code. Scala comes to help with its *mixin-class composition*. It allows the programmers to reuse the delta of a class definition, i.e., all new definitions that are not inherited. This mechanism makes it possible to combine `StringIterator` with `RichIterator`, as is done in the following test program which prints a column of all the characters of a given string.

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

The `Iter` class in function `main` is constructed from a mixin composition of the parents `StringIterator` and `RichIterator` with the keyword **`with`**. The first parent is called the *superclass* of `Iter`, whereas the second (and every other, if present) parent is called a *mixin*.

Operators

Any method which takes a single parameter can be used as an infix operator in Scala. Here is the definition of class `MyBool` which defines three methods `and`, `or`, and `negate`.

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

It is now possible to use `and` and `or` as infix operators:

```
def not(x: MyBool) = x negate; // semicolon required here
def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

As the first line of this code shows, it is also possible to use nullary methods as postfix operators. The second line defines an `xor` function using the `and` and `or` methods as well as the new `not` function. In this example the use of infix operators helps to make the definition of `xor` more readable.
Here is the corresponding code in a more traditional object-oriented programming language syntax:

```
def not(x: MyBool) = x.negate; // semicolon required here
def xor(x: MyBool, y: MyBool) = x.or(y).and(x.and(y).negate)
```

Packages

A package is a special object which defines a set of member classes, objects and packages. Unlike other objects, packages are not introduced by a definition.
A packaging **package** `p` *ds* injects all definitions in `ds` as members into the package whose qualified name is `p`. Members of a package are called *top-level definitions*. If a definition in `ds` is labeled private, it is visible only for other members in the package.

A protected modifier can be qualified with an package identifier `p` (e.g. `protected[`
Members labeled with such a modifier are also accessible from all code inside the package `p`.
Selections `p.m` from `p` as well as imports from `p` work as for objects. However, unlike other objects, packages may not be used as values. It is illegal to have a package with the same fully qualified name as a module or a class.
Top-level definitions outside a packaging are assumed to be injected into a special empty package. That package cannot be named and therefore cannot be imported. However, members of the empty package are visible to each other without qualification.
A compilation unit **package** `p;` *stats* starting with a package clause is equivalent to a compilation unit consisting of a single packaging **package** `p` *stats* .
Several packages may be declared in the same Scala source file:

```
package p1 {
  object test extends Application {
    Console.println("p1.test")
  }
}
package p2 {
  object test extends Application {
    Console.println("p2.test")
  }
}
```

Implicitly imported into every compilation unit are, in that order:

- the package `java.lang`,
- the package `scala`,
- and the object `scala.Predef`.

Members of a later import in that order hide members of an earlier import.

Pattern Matching

Scala has a built-in general pattern matching mechanism. It allows to match on any sort of data with a first-match policy.
Here is a small example which shows how to match against an integer value:

```
object MatchTest1 extends Application {
  def matchTest(x: Int): String = x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
  }
  Console.println(matchTest(3))
}
```

The block with the case statements defines a function which maps integers to strings. The `match` keyword provides a convenient way of applying a function (like the pattern matching function above) to an object.
Here is a second example which matches a value against patterns of different types:

```
object MatchTest2 extends Application {
  def matchTest(x: Any): Any = x match {
    case 1 => "one"
    case "two" => 2
    case y: Int => "scala.Int"
  }
  Console.println(matchTest("two"))
}
```

The first case matches if `x` refers to the integer value 1. The second case matches if `x` is equal to the string `"two"`. The third case consists of a typed pattern; it matches against any integer and binds the selector value `x` to the variable `y` of type integer.
Scala's pattern matching statement is most useful for matching on algebraic types expressed via case classes.
Scala also allows the definition of patterns independently of case classes, using `unapply` methods in extractor objects.

Polymorphic Methods

Methods in Scala can be parameterized with both values and types. Like on the class level, value parameters are enclosed in a pair of parentheses, while type parameters are declared within a pair of brackets.
Here is an example:

```
object PolyTest extends Application {
  def dup[T](x: T, n: Int): List[T] =
    if (n == 0) Nil
    else x :: dup(x, n - 1)
  Console.println(dup[Int](3, 4))
  Console.println(dup("three", 3))
}
```

Method `dup` in object `PolyTest` is parameterized with type `T` and with the value parameters `x: T` and `n: Int`. When method `dup` is called, the programmer provides the required parameters (see line 6 in the program above), but as line 7 in the program above shows, the programmer is not required to give actual type parameters explicitly. The type system of Scala can infer such types. This is done by looking at the types of the given value parameters and at the context where the method is called.

Regular Expression Patterns

Right-ignoring sequence patterns

Right-ignoring patterns are a useful feature to decompose any data which is either a subtype of `Seq[A]` or a case class with an iterated formal parameter, like for instance `Elem(prefix:String,label:String,attrs:MetaData,scp:NamespaceBinding, childre`
In those cases, Scala allows patterns having a wildcard-star `_*` in the rightmost position to stand for arbitrary long sequences.
The following example demostrate a pattern match which matches a prefix of a sequence and binds the rest to the variable `rest`.

```
object RegExpTest1 extends Application {
  def containsScala(x: String): Boolean = {
    val z: Seq[Char] = x
    z match {
      case Seq('s','c','a','l','a', rest @ _*) =>
        Console.println("rest is "+rest)
        true
      case Seq(_*) =>
        false
    }
  }
}
```

In contrast to previous Scala version, it is no longer allowed to have arbitrary regular expressions, for the reasons described below.

General RegExp patterns temporarily retracted from Scala

Since we discovered a problem in correctness, this feature is temporarily retracted from the Scala language. If there is request from the user community, we might reactivate it in an improved form.
According to our opinion regular expressions patterns were not so useful for XML processing as we estimated. In real life XML processing applications, XPath seems a far better option. When we discovered that our translation or regular expressions patterns has some bugs for esoteric patterns which are unusual yet hard to exclude, we chose it would be time to simplify the language.

Explicitly Typed Self References

When developing extensible software it is sometimes handy to declare the type of the value **this** explicitly. To motivate this, we will derive a small extensible representation of a graph data structure in Scala.
Here is a definition describing graphs:

```
abstract class Graph {
  type Edge
  type Node <: NodeIntf
  abstract class NodeIntf {
    def connectWith(node: Node): Edge
  }
  def nodes: List[Node]
  def edges: List[Edge]
  def addNode: Node
}
```

Graphs consist of a list of nodes and edges where both the node and the edge type are left abstract. The use of abstract types allows implementations of trait `Graph` to provide their own concrete classes for nodes and edges. Furthermore, there is a method `addNode` for adding new nodes to a graph. Nodes are connected using method `connectWith`.
A possible implementation of class `Graph` is given in the next program:

```
abstract class DirectedGraph extends Graph {
  type Edge <: EdgeImpl
  class EdgeImpl(origin: Node, dest: Node) {
    def from = origin
    def to = dest
  }
  class NodeImpl extends NodeIntf {
    def connectWith(node: Node): Edge = {
      val edge = newEdge(this, node)
      edges = edge :: edges
      edge
    }
  }
  protected def newNode: Node
  protected def newEdge(from: Node, to: Node): Edge
  var nodes: List[Node] = Nil
  var edges: List[Edge] = Nil
  def addNode: Node = {
    val node = newNode
    nodes = node :: nodes
    node
  }
}
```

Class DirectedGraph specializes the Graph class by providing a partial implementation. The implementation is only partial, because we would like to be able to extend DirectedGraph further. Therefore this class leaves all implementation details open and thus both the edge and the node type are left abstract. Nevertheless, class DirectedGraph reveals some additional details about the implementation of the edge type by tightening the bound to class EdgeImpl. Furthermore, we have some preliminary implementations of edges and nodes represented by the classes EdgeImpl and NodeImpl. Since it is necessary to create new node and edge objects within our partial graph implementation, we also have to add the factory methods newNode and newEdge. The methods addNode and connectWith are both defined in terms of these factory methods. A closer look at the implementation of method connectWith reveals that for creating an edge, we have to pass the self reference **this** to the factory method newEdge. But **this** is assigned the type NodeImpl, so it's not compatible with type Node which is required by the corresponding factory method. As a consequence, the program above is not well-formed and the Scala compiler will issue an error message.
In Scala it is possible to tie a class to another type (which will be implemented in future) by giving self reference this the other type explicitly. We can use this mechanism for fixing our code above. The explicit self type is specified after the constructor list (which is empty in our example below) with the keyword **requires**.
Here is the fixed program:

```
abstract class DirectedGraph extends Graph {
  ...
  class NodeImpl requires Node extends NodeIntf {
    def connectWith(node: Node): Edge = {
      val edge = newEdge(this, node)  // now legal
      edges = edge :: edges
      edge
    }
  }
  ...
}
```

In this new definition of class NodeImpl, this has type Node. Since type Node is abstract and we therefore don't know yet if NodeImpl is really a subtype of Node, the type system of Scala will not allow us to instantiate this class. But nevertheless, we state with the explicit type annotation of this that at some point, (a subclass of) NodeImpl has to denote a subtype of type Node in order to be instantiatable.
Here is a concrete specialization of DirectedGraph where all abstract class members are turned into concrete ones:

```
class ConcreteDirectedGraph extends DirectedGraph {
  type Edge = EdgeImpl
  type Node = NodeImpl
  protected def newNode: Node = new NodeImpl
```

```
  protected def newEdge(f: Node, t: Node): Edge =
    new EdgeImpl(f, t)
}
```

Please note that in this class, we can instantiate `NodeImpl` because now we know that `NodeImpl` denotes a subtype of type `Node` (which is simply an alias for `NodeImpl`). Here is a usage example of class `ConcreteDirectedGraph`:

```
object GraphTest extends Application {
  val g: Graph = new ConcreteDirectedGraph
  val n1 = g.addNode
  val n2 = g.addNode
  val n3 = g.addNode
  n1.connectWith(n2)
  n2.connectWith(n3)
  n1.connectWith(n3)
}
```

Subclassing

Classes in Scala are extensible. A subclass mechanism makes it possible to *specialize* a class by inheriting all members of a given *superclass* and defining additional class members.
Here is an example:

```
class Point(xc: Int, yc: Int) {
  val x: Int = xc
  val y: Int = yc
  def move(dx: Int, dy: Int): Point =
    new Point(x + dx, y + dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c
  def compareWith(pt: ColorPoint): Boolean =
    (pt.x == x) && (pt.y == y) && (pt.color == color)
  override def move(dx: Int, dy: Int): ColorPoint =
    new ColorPoint(x + dy, y + dy, color)
}
```

In this example we first define a new class `Point` for representing points. Then we define a class `ColorPoint` which extends class `Point`.
This has several consequences:

- Class `ColorPoint` inherits all members from its superclass `Point`; in our case, we inherit the values `x`, `y`, as well as method `move`.
- Subclass `ColorPoint` adds a new method `compareWith` to the set of (inherited) methods.
- Scala allows member definitions to be *overridden*; in our case we override the `move` method from class `Point`. This makes the `move` method of class `Point` inaccessible to clients of `ColorPoint` objects. Within class `ColorPoint`, the inherited method `move` can be accessed with a *super* call: `super.move(...)`. Opposed to Java where method overriding is *invariant* (i.e. the overriding method has to have the same signature), Scala allows methods to be overridden in a *contra/covariant* fashion. In the example above we make use of this feature by letting method `move` return a `ColorPoint` object instead of a `Point` object as specified in superclass `Point`.
- Subclasses define subtypes; this means in our case that we can use `ColorPoint` objects whenever `Point` objects are required.

For cases where we would like to inherit from several other classes, we have to make use of mixin-based class composition as opposed to pure subclassing.

Automatic Type-Dependent Closure Construction

Scala allows parameterless function names as parameters of methods. When such a method is called, the actual parameters for parameterless function names are not evaluated and a nullary function is passed instead which encapsulates the computation of the corresponding parameter (so-called *call-by-name* evalutation). The following code demonstrates this mechanism:

```
object TargetTest1 extends Application {
  def whileLoop(cond: => Boolean)(body: => Unit): Unit =
    if (cond) {
      body
      whileLoop(cond)(body)
    }
  var i = 10
  whileLoop (i > 0) {
    Console.println(i)
    i = i - 1
  }
}
```

The function `whileLoop` takes two parameters `cond` and `body`. When the function is applied, the actual parameters do not get evaluated. But whenever the formal parameters are used in the body of `whileLoop`, the implicitly created nullary functions will be evaluated instead. Thus, our method `whileLoop` implements a Java-like while-loop with a recursive implementation scheme.
We can combine the use of infix/postfix operators with this mechanism to create more complex statements (with a nice syntax).
Here is the implementation of a *loop-unless* statement:

```
object TargetTest2 extends Application {
  def loop(body: => Unit): LoopUnlessCond =
    new LoopUnlessCond(body)
  protected class LoopUnlessCond(body: => Unit) {
    def unless(cond: => Boolean): Unit = {
      body
      if (!cond) unless(cond)
    }
  }
  var i = 10
  loop {
    Console.println("i = " + i)
    i = i - 1
  } unless (i == 0)
}
```

The `loop` function just accepts a body of a loop and returns an instance of class `LoopUnlessCond` (which encapsulates this body object). Note that the body didn't get evaluated yet. Class `LoopUnlessCond` has a method `unless` which we can use as a infix operator. This way, we achieve a quite natural syntax for our new loop: `loop < stats > unless ( < cond > )`.
Here's the output when `TargetTest2` gets executed:

```
i = 10
i = 9
i = 8
i = 7
i = 6
i = 5
i = 4
i = 3
i = 2
i = 1
```

Traits

Similar to *interfaces* in Java, traits are used to define object types by specifying the signature of the supported methods. Unlike Java, Scala allows traits to be partially implemented; i.e. it is possible to define default implementations for some methods. In contrast to classes, traits may not have constructor parameterss.
Here is an example:

```
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}
```

This trait consists of two methods `isSimilar` and `isNotSimilar`. While `isSimilar` does not provide a concrete method implementation (it is *abstract* in the terminology of Java), method `isNotSimilar` defines a concrete implementation. Consequently, classes that integrate this trait only have to provide a concrete implementation for `isSimilar`. The behavior for `isNotSimilar` gets inherited directly from the trait. Traits are typically integrated into a class (or other traits) with a mixin class composition:

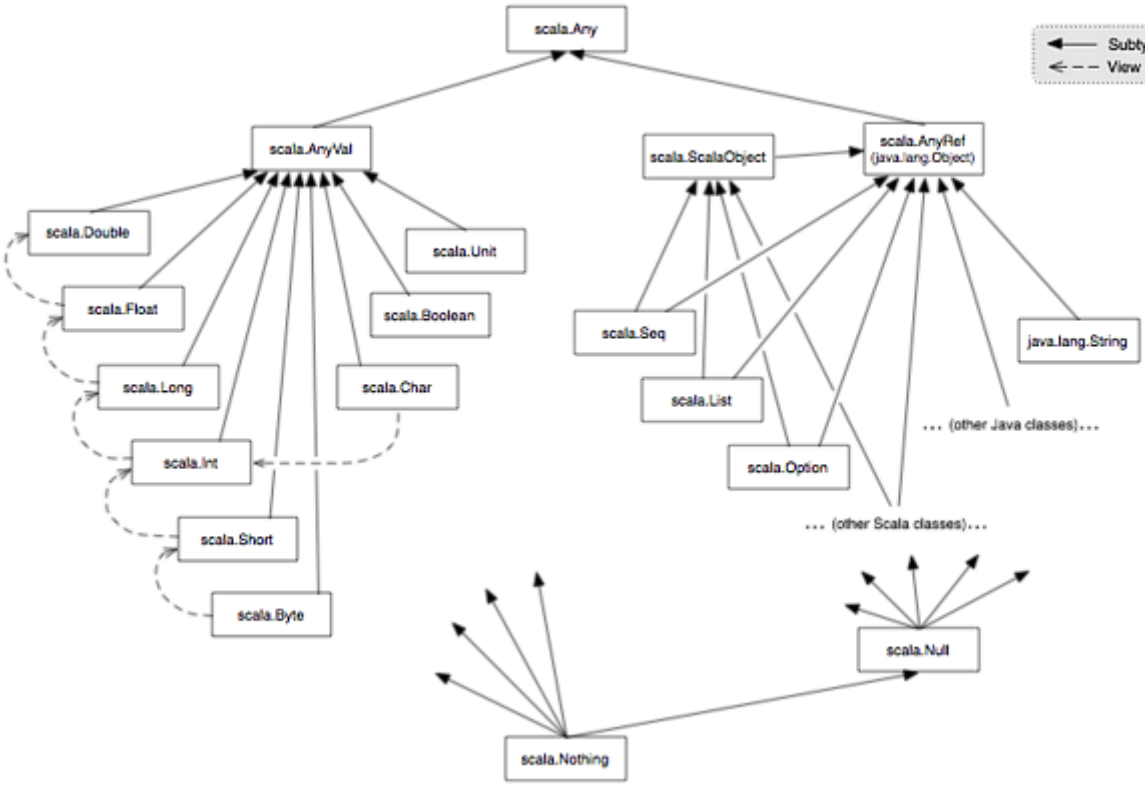
```
class Point(xc: Int, yc: Int) extends Similarity {
  var x: Int = xc
  var y: Int = yc
  def isSimilar(obj: Any) =
    obj.isInstanceOf[Point] &&
    obj.asInstanceOf[Point].x == x
}
object TraitsTest extends Application {
  val p1 = new Point(2, 3)
  val p2 = new Point(2, 4)
  val p3 = new Point(3, 3)
  Console.println(p1.isNotSimilar(p2))
  Console.println(p1.isNotSimilar(p3))
  Console.println(p1.isNotSimilar(2))
}
```

Here is the output of the program:

```
false
true
true
```

Unified Types

In contrast to Java, all values in Scala are objects (including numerical values and functions). Since Scala is class-based, all values are instances of a class. The diagram below illustrates the class hierarchy.



The superclass of all classes `scala.Any` has two direct subclasses `scala.AnyVal` and `scala.AnyRef` representing two different class worlds: *value classes* and *reference classes*. All value classes are predefined; they correspond to the primitive types of Java-like languages. All other classes define reference types. User-defined classes define reference types by default; i.e. they always (indirectly) subclass `scala.AnyRef`. Every user-defined class in Scala implicitely extends the trait `scala.ScalaObject`. Classes from the infrastructure on which Scala is running (e.g. the Java runtime environment) do not extend `scala.ScalaObject`. If Scala is used in the context of a Java runtime environment, then `scala.AnyRef` corresponds to `java.lang.Object`. Please note that the diagram above also shows implicit conversions called views between the value classs.
Here is an example that demonstrates that both numbers, characters, boolean values, and functions are objects just like every other object:

```
object UnifiedTypes {
  def main(args: Array[String]): Unit = {
```

```
    val set = new scala.collection.mutable.HashSet[Any]
    set += "This is a string"  // add a string
    set += 732                 // add a number
    set += 'c'                 // add a character
    set += true                // add a boolean value
    set += &main                // add the main function
    val iter: Iterator[Any] = set.elements
    while (iter.hasNext) {
      Console.println(iter.next.toString())
    }
  }
}
```

The program declares an application `UnifiedTypes` in form of a top-level singleton object with a `main` method. The `main` method defines a local variable `set` which refers to an instance of class `HashSet[Any]`. The program adds various elements to this set. The elements have to conform to the declared set element type `Any`. In the end, string representations of all elements are printed out.
Here is the output of the program:

```
c
true
<function>
732
This is a string
```

Upper Type Bounds

In Scala, type parameters and abstract types may be constrained by a *type bound*. Such type bounds limit the concrete values of the type variables and possibly reveal more information about the members of such types. An *upper type bound* `T <: A` declares that type variable `T` refers to a subtype of type `A`.
Here is an example which relies on an upper type bound for the implementation of the polymorphic method `findSimilar`:

```
trait Similar {
  def isSimilar(x: Any): Boolean
}
case class MyInt(x: Int) extends Similar {
  def isSimilar(m: Any): Boolean =
    m.isInstanceOf[MyInt] &&
    m.asInstanceOf[MyInt].x == x
}
object UpperBoundTest extends Application {
  def findSimilar[T <: Similar](e: T, xs: List[T]): Boolean =
    if (xs.isEmpty) false
    else if (e.isSimilar(xs.head)) true
    else findSimilar[T](e, xs.tail)
  val list: List[MyInt] = List(MyInt(1), MyInt(2), MyInt(3))
  Console.println(findSimilar[MyInt](MyInt(4), list))
  Console.println(findSimilar[MyInt](MyInt(2), list))
}
```

Without the upper type bound annotation it would not be possible to call method `isSimilar` in method `findSimilar`.
The usage of lower type bounds is discussed here.

Variances

Scala supports variance annotations of type parameters of generic classes. In contrast to Java 5 (aka. JDK 1.5), variance annotations may be added when a class abstraction is *defined*, whereas in Java 5, variance annotations are given by clients when a class abstraction is *used*.
In the page about generic classes an example for a mutable stack was given. We explained that the type defined by the class `Stack[T]` is subject to invariant subtyping regarding the type parameter. This can restrict the reuse of the class abstraction. We now derive a functional (i.e. immutable) implementation for stacks which does not have this restriction. Please note that this is an advanced example which combines the use of polymorphic methods, lower type bounds, and covariant type parameter annotations in a non-trivial fashion. Furthermore we make use of inner classes to chain the stack elements without explicit links.

```
class Stack[+A] {
  def push[B >: A](elem: B): Stack[B] = new Stack[B] {
    override def top: B = elem
    override def pop: Stack[B] = Stack.this
    override def toString() = elem.toString() + " " +
                              Stack.this.toString()
  }
  def top: A = error("no element on stack")
  def pop: Stack[A] = error("no element on stack")
  override def toString() = ""
}

object VariancesTest extends Application {
  var s: Stack[Any] = new Stack().push("hello");
  s = s.push(new Object())
  s = s.push(7)
  Console.println(s)
}
```

The annotation `+T` declares type *T* to be used only in covariant positions. Similarly, `-T` would declare *T* to be used only in contravariant positions. For covariant type parameters we get a covariant subtype relationship regarding this type parameter. For our example this means *Stack[T]* is a subtype of *Stack[S]* if *T* is a subtype of *S*. The opposite holds for type parameters that are tagged with a `-`.
For the stack example we would have to use the covariant type parameter `T` in a contravariant position for being able to define method `push`. Since we want covariant subtyping for stacks, we use a trick and abstract over the parameter type of method `push`. We get a polymorphic method in which we use the element type `T` as a lower bound of `push`'s type variable. This has the effect of bringing the variance of `T` in sync with its declaration as a covariant type parameter. Now stacks are covariant, but our solution allows that e.g. it's possible to push a string on an integer stack.

The result will be a stack of type *Stack[Any]*; so only if the result is used in a context where we expect an integer stack, we actually detect the error. Otherwise we just get a stack with a more general element type.

Views

Implicit parameters and methods can also define implicit conversions called *views*. A view from type `S` to type `T` is defined by an implicit value which has function type `S => T`, or by a method convertible to a value of that type.
Views are applied in two situations:

- If an expression `e` is of type `T`, and `T` does not conform to the expression's expected type `pt`.
- In a selection `e.m` with `e` of type `T`, if the selector `m` does not denote a member of `T`.

In the first case, a view `v` is searched which is applicable to `e` and whose result type conforms to `pt`. In the second case, a view `v` is searched which is applicable to `e` and whose result contains a member named `m`.
The following operation on the two lists `xs` and `ys` of type `List[Int]` is legal:

```
xs <= ys
```

assuming the implicit methods `list2ordered` and `int2ordered` defined below are in scope:

```
implicit def list2ordered[A](x: List[A])
    (implicit elem2ordered: a => Ordered[A]): Ordered[List[A]] =
  new Ordered[List[A]] { /* .. */ }

implicit def int2ordered(x: Int): Ordered[Int] =
  new Ordered[Int] { /* .. */ }
```

The `list2ordered` function can also be expressed witht he use of a *view bound* for a type parameter:

```
implicit def list2ordered[A <% Ordered[A]](x: List[A]): Ordered[List[A]] = .
```

The Scala compiler then generates code equivalent to the definition of `list2ordered` given above.
The implicitly imported object `scala.Predef` declares several predefined types (e.g. `Pair`) and methods (e.g. `error`) but also several views. The following example gives an idea of the predefined view `charWrapper`:

```
final class RichChar(c: Char) {
  def isDigit: Boolean = Character.isDigit(c)
  // isLetter, isWhitespace, etc.
}
object RichCharTest {
  implicit def charWrapper(c: char) = new RichChar(c)
```

```
  def main(args: Array[String]) {
    Console.println('0'.isDigit)
  }
}
```

XML Processing

Scala can be used to easily create, parse, and process XML documents. XML data can be represented in Scala either by using a generic data representation, or with a data-specific data representation. The latter approach is supported by the *data-binding* tool schema2src.

Runtime Representation

XML data is represented as labeled trees. Starting with Scala 1.2 (previous versions need to use the `-Xmarkup` option), you can conveniently create such labeled nodes using standard XML syntax.
Consider the following XML document:

```
<html>
  <head>
    <title>Hello XHTML world</title>
  </head>
  <body>
    <h1>Hello world</h1>
    <p><a href="scala.epfl.ch">Scala</a> talks XHTML</p>
  </body>
</html>
```

This document can be created by the following Scala program:

```
object XMLTest1 extends Application {
  val page =
  <html>
    <head>
      <title>Hello XHTML world</title>
    </head>
    <body>
      <h1>Hello world</h1>
      <p><a href="scala.epfl.ch">Scala</a> talks XHTML</p>
    </body>
  </html>;
  Console.println(page.toString())
}
```

It is possible to mix Scala expressions and XML:

```
object XMLTest2 extends Application {
  import scala.xml._
  val df = java.text.DateFormat.getDateInstance()
  val dateString = df.format(new java.util.Date())
  def theDate(name: String) =
  <dateMsg addressedTo={ name }>
```

```
    Hello, { name }! Today is { dateString }
  </dateMsg>;
  Console.println(theDate("John Doe").toString())
}
```

Data Binding

It many cases, you have a DTD for the XML documents you want to process. You will want to create special Scala classes for it, and some code to parse the XML, and to save. Scala comes with a nifty tool that turns your DTDs into a collection of Scala class definitions which do all of this for you.
Note that documentation and examples on the `schema2src` tool can be found in Burak's draft scala xml book.