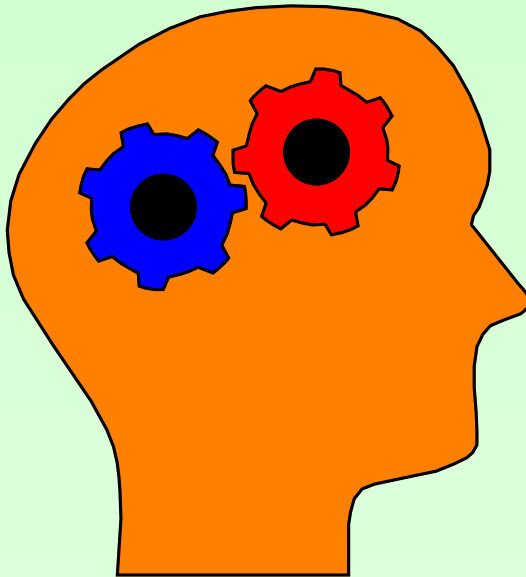




CS2104: Programming Languages Concepts

Lecture 12-13 : **Scala Highlights**



***“Boosting Java
With FP and Stronger Types”***

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Motivation for Scala Language

- interoperable with Java
- conciseness (2-10 times shorter)
- supports higher-order functions (OOP+FP)
- advance static typing and inference
- developed by Martin Odersky group @ EPFL

Conciseness

```
// in Java
class MyClass {
  private int index;
  private String name;
  public MyClass(int index, String name) {
    this.index = index;
    this.name = name;
  }
}
```

```
// in Scala:
class MyClass(index: Int, name: String)
```

Object-Oriented Style Supported

- Every value is an object
- Types and behavior of objects are described by
 - classes (including abstract classes)
 - *traits*
- Class abstraction are extended by
 - sub-classing
 - *mixin* composition
(cleaner replacement for multiple inheritance)

Functional Style Supported

- Every function is an object
- Functions are first-class values
 - passed as argument
 - returned as result
 - can be stored in data structures
- Anonymous functions allowed
- Pattern-matching via Case Classes

Highlights of Scala

- Scala Classes
- Types
- Higher-Order Functions
- Lists
- Pattern-Matching
- Traits as Mixins
- Implicits

Scala *Classes*

Scala Classes

- Factory templates that can be instantiated to objects.
- Class Parameters and Explicit Overriding

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


Scala Classes

- Parameterised by constructor arguments.
Objects are instantiated with new command, e.g.

```
new Point(3,4) ;
```

- Uses dynamically-dispatched methods only:

```
this.move(dx,dy) ;  
this.toString() ;
```

Scala Classes

- A class with a single object is declared with the "object" keyword.
- This example captures an executable application with a main method that can be directly executed.

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

Abstract Classes

- Classes are parameterized with values and with types.
- Supports generic classes.
- Abstract class may have
 - (i) deferred/abstract type
 - (ii) deferred value definition

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

Abstract Classes

- Can *reveal* more information on an abstract type by giving *type bounds*.

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

- *Refinement* could be added to instantiate abstract type definition:

```
abstract class IntSeqBuffer extends SeqBuffer {  
  type U = Int  
}
```

Abstract Classes

- Abstract type definition can be turned into type parameters with declaration-site variance annotation:

```
abstract class Buffer[+T] {  
  val element: T  
}
```

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

Types

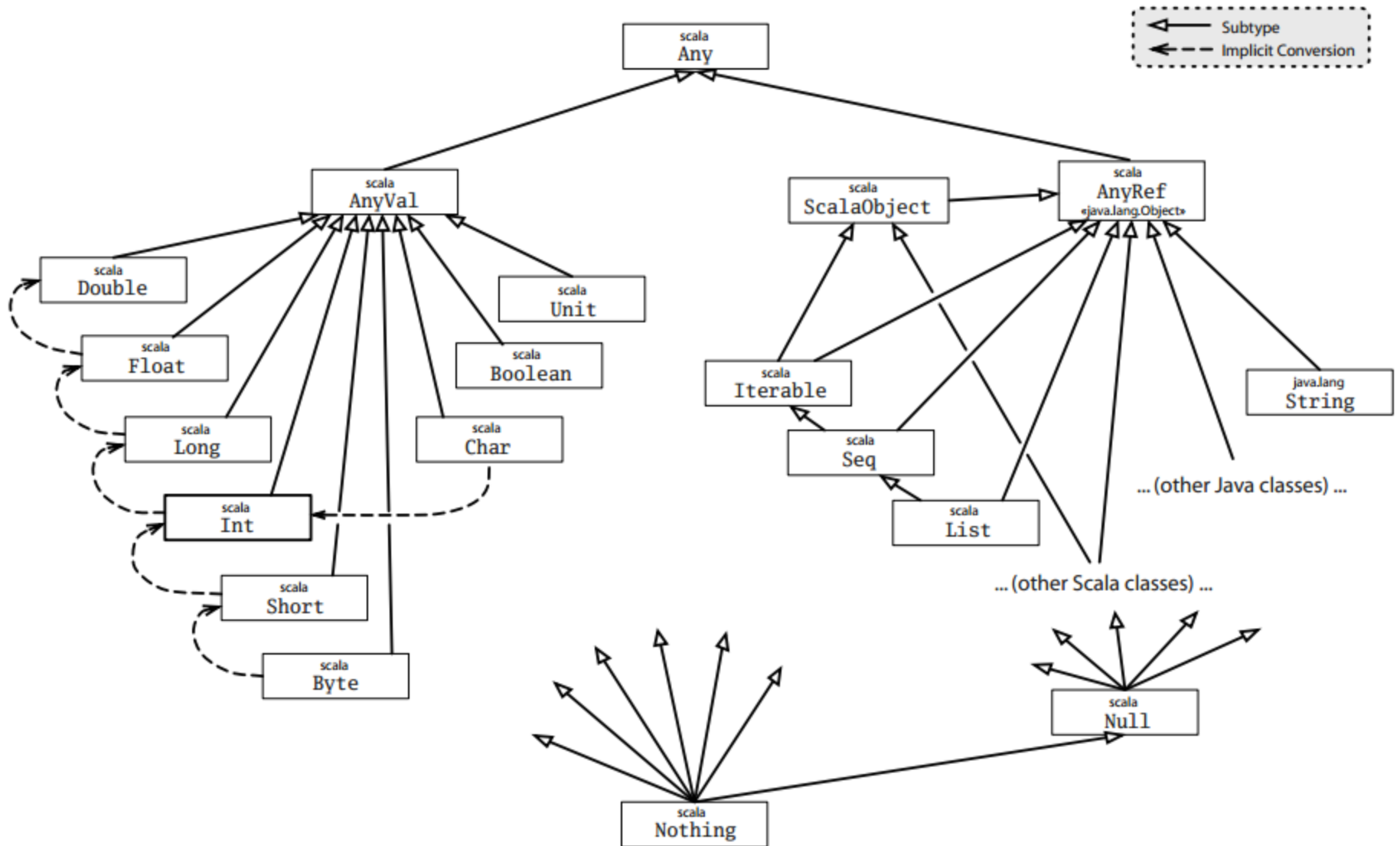


Figure 11.1 · Class hierarchy of Scala.

Unified Types

- all values (including numerics and functions) are objects
 - all values are instances of a class
 - superclass of all classes

scala.Any

:> scala.AnyVal

:> scala.AnyRef

- Every user-defined class
 - are (indirectly) subclass of **scala.AnyRef**
 - implicitly extends trait **scala.ScalaObject**

Unified Types

```
object UnifiedTypes {  
  def main(args: Array[String]) {  
    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.iterator  
    while (iter.hasNext) {  
      println(iter.next.toString())  
    }  
  }  
}
```

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

Polymorphic Methods

- Methods can be parameterized by both values and types
- Values are enclosed in parenthesis, while types are declared within a pair of brackets.

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

Lists

Pervasive List

- List is a pervasive data type in programming. Though Array also captures sequence, List has the following properties
 - (i) immutable
 - (ii) unbounded length
 - (iii) dynamically-linked data structure
- Examples:

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```

Pervasive List

- Two basic constructors (i) **Nil** (ii) **::**

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
  (0 :: (1 :: (0 :: Nil))) ::
  (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

- Three primitive List operations:
 - **head** returns the first element of a list
 - **tail** returns a list consisting of all elements except the first
 - **isEmpty** returns true if the list is empty

Pervasive List

Extractor is used to provide List patterns/views:

```
scala> val List(a, b, c) = fruit
a: String = apples
b: String = oranges
c: String = pears
```

```
scala> val a :: b :: rest = fruit
a: String = apples
b: String = oranges
rest: List[String] = List(pears)
```

Pervasive List

How would you *join* two Lists together?

```
def append[T](xs: List[T], ys: List[T]): List[T] =  
  xs match {  
    case List()    =>  
    case x :: xs1 =>  
  }
```

How would you *reverse* a given List?

```
def rev[T](xs: List[T]): List[T] =  
  xs match {  
    case List()    =>  
    case x :: xs1 =>  
  }
```

Folding over List

- Common to combine the elements of a list with some operator. For instance:

`sum(List(a, b, c)) equals 0 + a + b + c`

This summation can be implemented by:

```
def sum(xs: List[Int]): Int  
    = (0 /: xs) (_ + _)
```

- Similarly:

`product(List(a, b, c)) equals 1 * a * b * c`

This product operation can be implemented by:

```
def product(xs: List[Int]): Int = (1 /: xs) (_ * _)
```


Folding over Lists

- Fold left operation :

```
(z /: List(a, b, c)) (op)  
    equals op(op(op(z, a), b), c)
```

- Fold right operation:

```
(List(a, b, c) :\ z) (op)  
    equals op(a, op(b, op(c, z)))
```

Pattern Matching

Pattern Matching

- Match on any sort of data with a *first-match* policy
- Match keyword allows *pattern-matching function* to be applied to an object, e.g.

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

Pattern Matching

- Possible to match a value against patterns of different types. e.g.

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

Case Classes

- Case classes allow their constructor parameters to be exported via pattern-matching.
- Example to denote untyped lambda calculus:

```
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
```

Case Classes

- Advantages : "new" primitive is not required

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

- Constructor parameters are treated as public values that can be directly accessed

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

- Automatic derivation of *equality* and *toString* method.

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

Case Classes

- Supports pattern-matching:

```
// pretty printer
object TermTest extends Application {
  def printTerm(term: Term) {
    term match {
      case Var(n) =>
        print(n)
      case Fun(x, b) =>
        print("^" + x + ".")
        printTerm(b)
      case App(f, v) =>
        Console.print("(")
        printTerm(f)
        print(" ")
        printTerm(v)
        print(")")
    }
  }
}
```

Higher-Order Functions

Higher-Order Functions

- Scala supports functions as *first-class* values
 - function as parameter
 - function as result
 - function inside data structure.
- An example:

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

Higher-Order Functions

```
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("[", "]")  
  println(apply(decorator.layout, 7))  
}
```

- Polymorphic layout of type `A => String` is automatically coerced to a value of type `Int => String`

Anonymous Functions

- A shorthand for writing anonymous functions

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

- Full longer form :

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

- Function with two parameters

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

- Function with no parameter

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

Placeholder Function

- The symbol `_` denotes a placeholder for parameters

`_ + 1` or `(_:Int)+1`

denotes :

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

- Similarly:

`(_:Int) + (_:int)`

denotes :

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

Types for Functions

- Shorthand for Types:

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

- Longer form for Types

```
Function1[Int, Int]  
Function2[Int, Int, String]  
Function0[String]
```

Operators

- Infix and postfix form are occasionally more readable
 - Method with one parameter \rightarrow infix
 - Method with no parameter \rightarrow postfix.

```
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)  
}
```

Operators

- Possible to use "negate" in postfix form:

```
def not(x: MyBool) = x negate;  
    // semicolon required here
```

- Possible to use "and" and "or" in infix form:

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

- Traditional form:

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

Currying

- Methods may define multiple parameter lists.
- When a method is called with fewer argument list, it yields a function which expects the remaining parameter lists.

```
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)  
  println(filter(nums, modN(2)))  
  println(filter(nums, modN(3)))  
}
```

<pre>filter : (List[Int], Int => Boolean) => List[Int] modN : Int => (Int => Boolean)</pre>

Traits as Mixin

Traits

- Similar to interfaces in Java
 - can have fields and methods
 - may have default implementation from some methods
 - do not have constructor parameters

Example:

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

`isSimilar` is an abstract method,
but `isNotSimilar` has a concrete implementation

Traits

```
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)  
  println(p1.isNotSimilar(p2))  
  println(p1.isNotSimilar(p3))  
  println(p1.isNotSimilar(2))  
}
```

Mixin Class Composition

- Neither single inheritance, nor just multiple inheritance.
- Allows reuse of new member definitions from a class
- An abstract class:

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

Mixin Class Composition

- A mixin through keyword trait:

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

- A concrete iterator class (where type T has been instantiated):

```
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 += 1; ch }  
}
```

Mixin Class Composition

- A mixin class composition
(with both `StringIterator` and `RichIterator`)

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

Implicits

Implicit Parameters

- Implicit can help with type conversion.
- For example, `Double` cannot be automatically converted to `Int`.

```
scala> val i: Int = 3.5
<console>:4: error: type mismatch;
found   : Double(3.5)
required: Int
val i: Int = 3.5
```


Implicit Parameters

However, we can define an *implicit* conversion to automatically perform such casting.

```
scala> implicit def doubleToInt(x: Double)
      = x.toInt
doubleToInt: (x: Double)Int
scala> val i: Int = 3.5
i: Int = 3
```

Implicit Parameters

- Implicit allows our code to inter-operate with new types:

```
class Rational(n: Int, d: Int) {  
    ...  
    def + (that: Rational): Rational = ...  
    def + (that: Int): Rational = ...  
}
```

- Add an implicit to convert to Rational type:

```
implicit def intToRational(x: Int) =  
    new Rational(x, 1)
```

Implicit Parameters

- Default parameter can be customized using implicit.

```
class PreferredPrompt(val preference: String)
  object Greeter {
    def greet(name: String) (implicit prompt: PreferredPrompt) {
      println("Welcome, "+ name +". The system is ready.")
      println(prompt.preference)
    }
  }
```

Implicit parameter can be supplied explicitly:

```
scala> val bobsPrompt = new PreferredPrompt("relax> ")
scala> Greeter.greet("Bob") (bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```

Implicit Parameters

- We can also define it implicitly:

```
object JoesPrefs {  
  implicit val prompt =  
    new PreferredPrompt("Yes, master> ")  
}
```

- This results in :

```
scala> import JoesPrefs._  
import JoesPrefs._  
scala> Greeter.greet("Joe")  
Welcome, Joe. The system is ready.  
Yes, master>
```

Implicit Parameters

- a special feature to support systematic method overloading
- define abstract classes

```
abstract class SemiGroup[A] {  
  def add(x: A, y: A): A  
}  
abstract class Monoid[A] extends SemiGroup[A] {  
  def unit: A  
}
```

Implicit Parameters

- allow instances of these abstract classes.

e.g. `Int` and `String` are indirectly instances of `Monoid[?]`

```
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  
  }  
}
```

Implicit Parameters

- with implicit objects, we can now define generic method, that sum up a list of monoid values:

e.g.

```
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =  
  if (xs.isEmpty) m.unit  
  else m.add(xs.head, sum(xs.tail))
```

- implicit parameters can be inferred
given:

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

can infer:

```
println(sum(List(1, 2, 3))(IntMonoid))  
println(sum(List("a", "b", "c"))(StringMonoid))
```

Miscellaneous

Packages

- A package is a special object which defines a set of member classes, objects and packages.
- A packaging package $p \{ ds \}$ injects all definitions in ds as members into the package whose qualified name is p .
- If a definition in ds is labeled *private*, it is visible only for other members in the package.
- A *protected* modifier allows its members to be accessible from all code inside the package p .

Import

- An import clause determines a set of names of that can be used without qualifications.

```
import p._  
  //all members of p  
  //(this is analogous to import p.* in Java).  
import p.x  
  //the member x of p.  
import p.{x => a}  
  //the member x of p renamed as a.  
import p.{x, y}  
  //the members x and y of p.  
import p1.p2.z  
  //the member z of p2, itself member of p1.
```

Import

- Implicitly imported into every compilation unit:
 - the package `java.lang`,
 - the package `scala`,
 - and the object `scala.Predef`.

Type Inference

Local Type Inference

- Types can either be declared or inferred.
- Eases programmers burden by automatically inferring certain type annotations.
- Can infer type of:
 - (i) variable (through its initialization)
 - (ii) results of non-recursive method
 - (iii) type instantiation of polymorphic methods
- May fail occasionally.

Local Type Inference

Example 1: Type Instantiation of Generic Methods

```
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  
}
```

Explicit Instantiation:

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

Local Type Inference

Example 2:

```
object InferenceTest2 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  
}
```

Failures of type inference

```
object InferenceTest3 {  
  def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)  
}  
  
object InferenceTest4 {  
  var obj = null // Null inferred  
  obj = new Object()  
}
```

Runtime Type Representation

- `classOf[T]` returns string representation of a type
- `var.getClass()` returns the representation of runtime type for object

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