**Natural Language Processing: Fall 2013**

**NLP Class**

- Home
- Syllabus
- Schedule
- Notes
- Assignment Requirements
- Links

**Useful Information**

- Scala

**Assignments**

- #0 - Programming
- #1 - Probability
- #2 - Classification
- #3 - N-Grams
- #4 - HMMs
- #5 - MaxEnt
- #6 - Parsing

**External Links**

- UTCL Main site
- Blackboard

# Scala Basics



## Variables

There are two keywords for declaring variables: `val` and `var`. Identifers declared with `val` cannot be reassigned; this is like a `final` variable in Java. Identifers declared with `var` may be reassigned.

```
val a = 1
var b = 2

b = 3   // fine
a = 4   // error: reassignment to val
```

You should (pretty much) exclusively use `val`. If you find yourself wanting to use a `var`, there is almost certainly a better way to structure your code.

## Style

- class names start with a capital letter
- variables and methods start with lowercase letters
- constants start with capitals
- everything uses camelCase

## Specifying Types

Scala has powerful type inference capabilities, so, in many cases, types do not need to be specified. However, types may be specified at any time. This is often useful for making complex code more readable, or as a way of protecting against errors. Additionally, types can be specifed on any subexpression, not just on variable assignments.

```
val a = 4                  // a: Int = 4
val b: Int = 4             // b: Int = 4
val c = (a: Double) + 5    // c: Double = 9.0
```

All types are determined statically during compilation.

## Basic Syntax

### Imports

Classes, objects, and static methods can all be imported. An underscore can be used as a wildcard to import everything from a particular context.

```
import scala.collection.immutable.BitSet
import scala.math.log
import scala.math._
```

### Semi-colon Inference

Scala will infer semi-colons at the ends of lines, so they do not need to be explicitly written. If you add them yourself, then Scala will not complain, but it doesn't look as nice.

Semi-colons can also be used to write several statements on a single line, but this should be avoided because it's harder to read in most circumstances.

### String Interpolation

```
val x = 4
val y = "stuff"
val z = f"You can say $x is inverse of ${1.0 / x}%.2f and $y"
```

See the docs for more.

### Control

Scala has control many familiar control structures.

#### if-else

```
val x = 4
if(x > 2)
  println("greater than 2")
else if(x < 4)
  println("less than to 2")
else
  println("equal to 2")
// prints "greater than 2"
```

#### for-each loop

```
val xs = Vector(1,2,3,4,5)
for(x <- xs)
  println(x)
// prints numbers 1 through 5
```

But the for-each loop can be used in more complex ways, allowing succinct syntax for looping over multiple collections and filtering:

```
for(
   x <- Vector(1,2,3,4,5);  // outer loop over a vector
   if x % 2 == 1;           // filter out even xs
   y <- Set(1,2,3);         // inner loop over a list
   if x + y == 6            // filter out entries that don't sum to 6
  ) println(s"x=$x, y=$y")
// prints:
//   x=3, y=3
//   x=5, y=1
```

This is equivalent to:

```
for(x <- Vector(1,2,3,4,5))
  if(x % 2 == 1)
    for(y <- Set(1,2,3))
      if(x + y == 6)
        println(s"x=$x, y=$y")
```

### Everything is an Expression

In Scala, many things are expression that are not in other languages.

Blocks are expressions that are evaluated and resolve to the value of the final expression in the block:

```
val x = {
  val intermediate1 = 2 + 3
  val intermediate2 = 4 + 5
  intermediate1 * intermediate2  // will be "returned" from the block
}
// x: Int = 45
```

If-else constructs are expressions whose value is the branch that is taken. The return type of an if-else expression is the lowest common ancestor of the values of each branch.

```
val a = 4
val x =                // type is inferred as Iterable[Int]
  if(a > 2)            //    because
    Vector(1,2,3)      //      Vector extends Iterable
  else                 //    and
    Set(4,5)           //      so does Set
// x: Iterable[Int] = Vector(1, 2, 3)
```

## Functions

Functions are defined using the `def` keyword.

A few points:

- Parameter types must be specified.
- There can be multiple parameter lists
- Return types are optional: they can be inferred at compile-time. (Unless the function is recursive.)
- The body of the function should be separated from the signature by an equals sign (unless the return type is `Unit`, indicating no return value – a “void” function). This keeps the syntax consistent with assignments: name on the left, expression on the right.
- Braces are not needed around the function body if it is only a single expression. The equals sign must be followed by a single expression, but, as discussed above, expressions can take many forms including brace-enclosed blocks.
- Parentheses are not needed in the function signature if there are no parameters. If the function has empty parentheses, then they are optional on the call. If the function is defined without parentheses, then they are not allowed, making the call look like a variable access, except that the value is recomputed on every access (“uniform access principle”).
- The `return` keyword is not needed (and should be avoided). Since every block is an expression, and the last expression in a block is the value of the block, the result of the last expression in a function body will be the returned value.

Some examples

```
def add(i: Int, j: Int) = i + j     // no braces needed
def add2(i: Int)(j: Int) = i + j    // two parameter lists
def mystring() = "something"        // parentheses option in caller
def mystring2 = "something else"    // no parentheses allowed in
caller
def doubleSum(i: Int, j: Int) = {   // braces for multiple statements
  val sum = i + j
  sum * 2                           // "return value"
}
def ceilHalf(n: Int) = {
  if(n % 2 == 0)                    // if-else expression is the final
    n / 2
  else
    (n + 1) / 2
}
def mult(i: Int, j: Int): Int = i * j  // return type specified
```

```
add(2,3)                 // res48: Int = 5
add2(2)(3)               // res49: Int = 5
mystring()               // res50: String = something
mystring                 // res51: String = something
mystring2                // res52: String = something else
doubleSum(2,3)           // res53: Int = 10
ceilHalf(3)              // res54: Int = 2
mult(2,3)                // res55: Int = 5
```

## Classes

Classes can be declared using the `class` keyword. Methods are declared with the `def` keyword. Methods and fields are public by default, but can be specified as `protected` or `private`. Constructor arguments are, by default, private, but can be proceeded by `val` to be made public.

```
class A(i: Int, val j: Int) {
  val iPlus5 = i + 5
  private[this] val jPlus5 = j + 5

  def addTo(k: Int) = new A(i + k, j + k)
  def subtractFrom(k: Int): (Int, Int) = (i + k, j + k)
  def sum = i + j
  def doSomeStuff() = {
    val a = i + jPlus5
    val b = j - i + jPlus5
    (a, b)
  }
}
val a = new A(2,3)
a.j                 // accessing a public constructor-arg field
a.iPlus5            // accessing a public field
a.addTo(6)          // calling a method with an argument
a.sum               // calling a no-arg method, parentheses not
permitted
a.doSomeStuff       // calling a no-arg method, parentheses optional
```

### Inheritance

Classes are extended using the `extends` keyword

```
class B(i: Int, k: Int) extends A(i, 4)
```

### Traits

Traits are like interfaces, but they are allowed to have members declared (“mix-in” members).

```
trait C {
  def methodToImplement: Int
  def doCThing = "C thing"
}

trait D {
  def doDThing = "D thing"
}

class E extends C with D {
  override def methodToImplement = 7
}
val e = new E          // e: E = E@766b0524
e.doCThing             // res14: String = C thing
```

### Objects

Scala does not allow static members of classes or traits. Instead all static members must be declared on an `object`.

```
object F {
  val num = 5
  def add(x: Int, y: Int) = x + y
}

F.num          // res15: Int = 5
F.add(3, 4)    // res16: Int = 7
```

The terminology is somewhat confusing since an “object” can also mean an instantiated instance of a class.

### Case Classes

Case classes are syntactic sugar for classes with a few methods pre-specified for convenience. These include `toString`, `equals`, and `hashCode`, as well as static methods `apply` (so that the `new` keyword is not needed for construction) and `unapply` (for pattern matching). Case class constructor args are also public by default. Case classes are not allowed to be extended. Otherwise, they are just like normal classes.

```
case class G(i: Int, j: Int) {
  def sum = i + j
}

val g = G(4, 5)     // g: G = G(4,5)
g.sum               // res19: Int = 9
g == G(4,5)         // res21: Boolean = true
```

### Operators (or lack thereof)

Scala does not have operators. Anything that looks like an operator in Scala is actually a method:

```
scala> "this" + "that"
res0: String = "thisthat"

scala> "this".+("that")
res1: String = thisthat
```

Scala just knows that if there is no dot or parentheses, then it should treat the expression as a call to a 1-argument method called +.

You can, therefore, define your own “operators”:

```
case class A(i: Int) {
  def +(a: A) = A(i + a.i)
  def ++-||-++(j: Int) = A(i - j)
}
```

Which can be used like this:

```
scala> A(5) + A(2)
res0: A = A(7)

scala> A(5) ++-||-++ 2
res1: A = A(3)
```

However, **do not abuse this power**. It can make your code extremely unreadable.

### Dot-and-Parentheses Dropping

Since Scala makes no distinction between methods and “operators”, you can actually drop the dot and parentheses from any 1-argument method:

```
case class A(i: Int) {
  def addTo(a: A) = A(i + a.i)
}
```

```
scala> A(5) addTo A(2)
res0: A = A(7)
```

This can *sometimes* make things more readable:

```
scala> 1 to 5
res1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
```

Again, **do not abuse this power** or your code will become extremely unreadable.

I really dislike overuse of this notation. I think it is appropriate in limited circumstances where the two sides of the operator have something like equal standing, like in the example with `to` for a Range where it's not quite fair to call the end of a range an argument, even though it technically is.

## Tuples

Scala has Tuple types for 1 though 22 elements. In a tuple, each element has its own type, and each element can be accessed using the `._n` syntax, where `n` is a 1-based index.

```
scala> val a = (1, "second", 3.4)
a: (Int, String, Double) = (1,second,3.4)

scala> a._2
res0: String = second
```

Just for fun, Scala has the method `->` defined on `Any` (and is therefore inherited by all types) for constructing a `Tuple2`:

```
scala> 1 -> 2
res1: (Int, Int) = (1,2)
```

## Collections

The Scala collections framework is pretty extensive. But for now, I'll just introduce the three most important collections:

### Vector

A `Vector[T]` is a sequence of items of type `T`. Elements can be accessed by 0-based index.

```
scala> val a = Vector(1,2,3)
a: Vector[Int] = Vector(1, 2, 3)

scala> a(0)
res0: Int = 1
```

### Set

A `Set[T]` is an unordered collection of items of type `T`. Since it's a set, no element can appear more than once. Since there is no order in a Set, elements cannot be accessed by index, but it is possible to check whether an element is in the set.

```
scala> val a = Set(1,2,3,2,3)
a: Set[Int] = Set(1, 2, 3)

scala> a(1)
res1: Boolean = true
```

### Map

A `Map[K,V]` is an associative array or dictionary type mapping elements of type `K` to elements of type `V`. Values can be accessed through their keys.

```
scala> val a = Map(1 -> "one", 2 -> "two", 3 -> "three")
a: Map[Int,String] = Map(1 -> one, 2 -> two, 3 -> three)
```

```
scala> a(1)
res2: String = one
```

Note: Remember that the syntax `a -> b` is nothing more than writing the pair `(a,b)`.

### Iterator

An `Iterator[T]` is a lazy sequence, meaning that it only evaluates its elements once they are accessed. Additionally, iterators can only be traversed one time. This is very useful for things like conserving memory by handling one element at a time or saving time by only evaluating as many elements as you need.

Accidentally traversing the same iterator more than once is a common source of bugs. If you want to be able to access the elements more than once, you can always call `.toVector` to load the entire thing into memory.

```
val a = Iterator(1,2,3)
val b = a.map(x => x + 1) // stage an operation, but don't traverse
yet
val c = b.sum             // c: Int = 9
val d = b.mkString(" ")   // d: String = ""

val e = Iterator(1,2,3)
val f = e.map(x => x + 1) // stage an operation, but don't traverse
yet
val g = f.toVector        // g: Vector[Int] = Vector(2, 3, 4)
val h = g.sum             // h: Int = 9
val i = g.mkString(" ")   // i: String = "2 3 4"
```

## Pattern Matching

Pattern matching is a really awesome capability of Scala. It's extremely useful and flexible, allowing you to write very succinct code. It can be used in a variety of situations, and many built-in Scala types already have pattern-matching behavior defined.

Variable assignment:

```
val a = (1,2)
val (x,y) = a          // x: Int = 1, y: Int = 2

val b = Vector(1,2)
val Vector(x,y) = b    // x: Int = 1, y: Int = 2

val c = Some(5)
val Some(x) = c        // x: Int = 5
```

Match expressions:

```
val a = Vector(1,2,3)

val sum =                              // sum: Int = 6
  a match {
    case Vector(x,y) => x + y
    case Vector(x,y,z) => x + y + z
  }
```

For-loops:

```
val a = Vector((1,2), (3,4), (5,6))
for((x,y) <- a)         // prints 3, 7, and 11
  println(x + y)
```

Anonymous (partial) functions. More on this example later. Note the use of curly braces instead of parentheses.

```
val a = Vector((1,2), (3,4), (5,6))
```

```
a.map { case (x,y) => x + y }  // res0: Vector[Int] = Vector(3, 7, 11)
```

In all of these cases, the matching function works the same way.

### Matching Regular Expressions

You can match directly with regular expressions:

```
val SomeRE = """a+b+""".r
"aaabb" match {
  case SomeRE() => "match found!"
}
// match found!
```

Additionally, by using parentheses, you can indicate groups in the pattern that are to be captured, and then have the pattern matcher assign those captured groups to variables:

```
val SomeRE = """(\d+), (\S+) \S+ (\S+).*""".r
"12, two more words plus some other stuff" match {
  case SomeRE(a, b, c) => f"matches with a=$a b=$b c=$c"
}
// matches with a=12 b=two c=words
```

### Constants, Wildcards, Sequences, Conditions, and Recursive Matching

Pattern matching is very flexible and allows not just for matching flat collections of variables.

A pattern can contain constants. In order for the matcher to recognize these terms as constants and not variables, they must either be a literal, start with a capital letter, or appear in backticks:

```
val C = 2
val v = 3
(1,2,3) match {
  case (1, C, `v`) => "this will match"
}
```

Wildcards are useful for specifying that a portion of the expression can match anything. This is often used as a “default” case.

```
(1,2,3) match {
  case (1, _, _) => "this will match anything that starts with a 1"
  case _ => "will match anything not matched by an above case"
}
```

Sequences can be matched without knowing exactly how many items there are:

```
Vector(1,2,3,4) match {
  case Vector(x, y, _*) => "matches with x=1, y=2, and ignores rest"
}
```

Conditions can be specified with `if` clauses:

```
(1,2) match {
  case (x, y) if x == y => "this will match if x == y"
}
```

Patterns can be nested for more complex matching:

```
Vector((1,2), (3,4), (5,6)) match {
  case Vector((x, y), _*) => "this will match with x=1, y=2"
}
```

### Variable Binding of a Pattern

It is also possible to match a pattern and then bind the matched portion to a variable. This is done with the `@` symbol:

```
Vector((1,2), (3,4), (5,6)) match {
```

```
    case a @ Vector((x, y), _*) => "binds entire Vector to `a`"
    case Vector(a @ (x, y), _*) => "binds first pair to `a`"
    case Vector((x, y), a @ _*) => "binds tail sequence to `a`"
  }
```

### Case Classes

Case classes automatically specify the behavior required for use in pattern matching.

```
case class A(i: Int, j: Int)
val a = A(1,2)
a match {
  case A(x,y) => "this will match with x=1, y=2"
}
```

### Defining Extractors

Many Scala types come with pattern matching behavior defined, as does any class defined as a case class. However, it is possible to define arbitrary pattern matching behavior for your own situations. Matching behavior is defined by the `unapply` method on either a class or object.

```
object Half {
  def unapply(n: Int) =
    if(n % 2 == 0)
      Some((n/2, n/2)) // indicates a match, specifies return
behavior
    else
      None             // indicates no match
}

Vector(1,2,3).map {
  case Half(x,y) => s"match with ($x,$y)"
  case _ => "no match"
}
// res0: Vector[String] = Vector(no match, match with (1,1), no
match)
```

This is, in fact, how pattern matching is implemented for all Scala built-in classes as well. For example, there is a `Vector` object that has an `unapply` method. And for every case class that is defined, the compiler generates an object with the same name and implements an `unapply` method on it.

Extractors for variable-length patterns (like `Vector` has) can be specified with an `unapplySeq` method.

## Functional Programming

### Favor Immutability

You should (pretty much) always use immutable collections. You code will therefore consist largely of operations on collections that produce new collections. Immutability keeps your code safer, makes it easier to reason about what is happening, and can have performance gains when used correctly.

### First-class functions

One of the most important characteristics of functional programming is that functions are first-class members of the language. This means that they can be stored in variables and, more importantly, passed as arguments to other functions.

To facilitate these kinds of uses, Scala has nice syntax for defining anonymous functions. In Scala, the symbol => is used to write lambda functions:

```
val add1a = (x: Int) => x + 1          // arg type declared
val add1b: (Int => Int) = x => x + 1   // function's type
```

```
declared
add1a(2)                                    // res0: Int = 3
add1b(2)                                    // res1: Int = 3

def addSome(f: (Int => Int), i: Int) = f(i) // first arg is a
function
addSome(x => x + 1, 2)                      // res2: Int = 3
addSome(add1a, 2)                           // res3: Int = 3
```

Scala also provides the ability to write an underscore (_) as short-hand for `x => x` (kind of).

```
val add2a: (Int => Int) = _ + 2          // function's type declared
val add2b: (Int => Int) = 2 + _          // function's type declared
add2a(2)                                 // res4: Int = 4
add2b(2)                                 // res5: Int = 4
addSome(_ + 2, 2)                        // res6: Int = 4
addSome(add2a, 2)                        // res7: Int = 4
```

### Some fundamental methods

The Scala API is extremely useful for finding out what methods are availble on various collections. There are a lot of methods, but here are a few of the most important:

**map**: Take a function as an argument and apply it to every element in the collection.

```
Vector(1,2,3).map(x => x + 2)   // same as...
Vector(1,2,3).map(_ + 2)        // res0: Vector[Int] = Vector(3, 4, 5)
```

**flatten**: Flatten a collection of collections.

```
val a = Vector(Vector(1,2,3), Vector(4,5,6), Vector(7,8,9))
a.flatten   // res1: Vector[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

**flatMap**: Map a function over the collection and flatten the result

```
Vector(1,2,3).flatMap(n => Vector.fill(n)(s"[$n]"))
// res2: Vector[String] = Vector([1], [2], [2], [3], [3], [3])
```

**foldLeft**: Map a function over the collection, accumulating the results. Takes two parameters: the base value and the function.

```
Vector(1,2,3).foldLeft(0)((accum, x) => accum + x)    // res3: Int = 6
```

**filter**: Remove items for which the given predicate is false

```
Vector(1,2,3).filter(x => x % 2 == 1)    // same as...
Vector(1,2,3).filter(_ % 2 == 1)    // res4: Vector[Int] = Vector(1,
3)
```

Worth noting: Scala is extremely clever about giving you back the type of collection that you'd expect. It tries to give you back what you started with, and if it can't, then it gives you the closest thing possible.

### For-Comprehensions

In Scala, the for-statement is actually syntactic sugar for a series of calls to collections methods.

When you write the statement:

```
for(
   x <- Vector(1,2,3,4,5);
   y <- Set(1,2,3)
  ) println(x + y)
```

the compiler rewrites this as:

```
Vector(1,2,3,4,5).foreach(x =>
  Set(1,2,3).foreach(y =>
    println(x + y)))
```

Since these loops do not evaluate to anything, they can be thought of as statements. (Technically they are expressions that evaluate to `Unit`, but that's the same as not evaluating to anything.)

Scala also provides a mechanism for for-comprehensions, expressions with similar syntax that produce collections. These are signaled by the `yield` keyword:

```
val a =
  for(
     x <- Vector(1,2,3,4,5); // outer loop over a vector
     if x % 2 == 1;          // filter out even xs
     y <- Set(1,2,3);        // inner loop over a list
     if x + y == 6           // filter out entries that don't sum to
6
    ) yield x * y
// a: scala.collection.immutable.Vector[Int] = Vector(9, 5)
```

This is accomplished by having the compiler translate the expression into a series of method calls. Each iteration is a call to `flatMap` and the last iteration is a call is to `map`:

```
val a =
  Vector(1,2,3,4,5)
    .filter(x => x % 2 == 1)
    .flatMap(x =>
      Set(1,2,3)
        .filter(y => x + y == 6)
        .map(y => x * y))
// a: scala.collection.immutable.Vector[Int] = Vector(9, 5)
```

### Parallelization

The many of the higher-order functions on Scala's collections can trivially be run in parallel, and because they are immutable, they are implicitly thread-safe.

Collections are converted to their parallel versions with the method `.par`. Parallel collections can be converted back using `.seq`. For example:

```
val a = Vector(1,2,3,4)      // create a vector
val b = a.par                // make the vector parallel
val c = b.map(x => x + 1)    // functions executed on different cores
val d = c.seq                // back to a normal vector
```

Parallelization has additional overhead, but with very large collections and many cores, it will likely be much faster than sequential execution.

Note that not all operations are parallelizable. For example, `foldLeft` cannot be parallelized because it must be run left-to-right. If you want to fold in a parallelizable way, you must use `fold` and give it a function that can be run out of order:

```
val a = Vector(1,2,3,4).par.fold(0)((x,y) => x + y)
```

## Use Option not null

Don't ever use `null`. Ever.

Scala provides a *much* better alternative: `Option`. An `Option` is basically a box around a type that can either contain an object of that type, or contain nothing. This is implemented such that `Option[T]` is a trait (interface), and it has two implementing classes: `Some[T]` and `None`.

So, if you write a function that, for example, needs to return an `Int`, but that might sometimes not want to return a value, instead of returning `null` as the default, you can make the return type `Option[Int]`, and return either `Some[Int]`, when there is a value, or `None`, when there is not:

```
def indexOf[T](a: T, xs: Vector[T], i: Int = 0): Option[Int] = {
  if(i >= xs.length)
    None                         // Vector exhausted.  No match found.
  else if(xs(i) == a)
```

```
      Some(i)                     // Found a match.  Return the index.
    else
      indexOf(a, xs, i + 1)       // Not a match.  Keep looking.
  }

  indexOf('c', "abcdefg".toVector)     // res0: Option[Int] = Some(2)
  indexOf('k', "abcdefg".toVector)     // res1: Option[Int] = None
```

When we get the result, we will always know whether the value is present or not, *without having to null-check*. Furthermore, `Option` has a number of really terrific methods analogous to those on the collections, that make using an `Option` very easy.

For example, if you wanted to perform some action on the result of `indexOf`, but only if there was actually a result, in Java you might do something like this:

```
// DON'T DO THIS!!!
val n: Int = indexOfWithNull('c', "abcdefg".toVector)
val s: String =
  if(n != null)
    s"the index was $n!"
  else
    null
  }
if(s != null)
  println(s)
```

But in Scala you can do this:

```
val s: Option[String] =
  indexOf('c', "abcdefg".toVector) match {
    case Some(n) => Some(s"the index was $n!")
    case None => None
  }
s match {
  case Some(x) => println(x)
  case None =>
}
```

That keeps things nicely within the `Option` world so that there is no question about whether any value might or might not be `null` at any time, and when null-checks are necessary. However, it's still a bit verbose. We can get the same result by doing this:

```
indexOf('c', "abcdefg".toVector)
  .map(n => s"the index was $n!")
  .foreach(println(_))
```

So see examples of all the various methods on `Option`, see Scala Option Cheat Sheet.

As a final note, `Option` is really great for chaining together operations that should only succeed if all the values are present, and fall to `None` if any is `None`. If we were looking values up in a `Map`, we could do something like:

```
val m = Map(1 -> "one", 2 -> "two", 3 -> "three")
val s1 =                     // s1: Option[String] = Some("onetwothree")
  for(
    a <- m.get(1);
    b <- m.get(2);
    c <- m.get(3)
  ) yield (a + b + c)

val s2 =                     // s2: Option[String] = None
  for(
    a <- m.get(1);
    b <- m.get(4);
    c <- m.get(3)
  ) yield (a + b + c)
```

## Implicit Classes

Scala allows you to "add" behavior to existing classes in a principled way using implicit classes. An implicit class takes exactly one constructor argument that is the type to be extended and defines behavior that should be allowed for that type.

```
implicit class EnhancedVector(xs: Vector[Int]) {
  def sumOfSquares = xs.map(x => x * x).sum
}

Vector(1,2,3).sumOfSquares        // res0: Int = 14
```

## Magic methods

### apply

The `apply` method of a class or object is used to overload the parentheses syntax, allowing you to specify the behavior of what looks like function application.

```
class A(i: Int){
  def apply(j: Int) = i + j
}

val something = new A(3)
something(4)                        // res0: Int = 7
```