

Functions

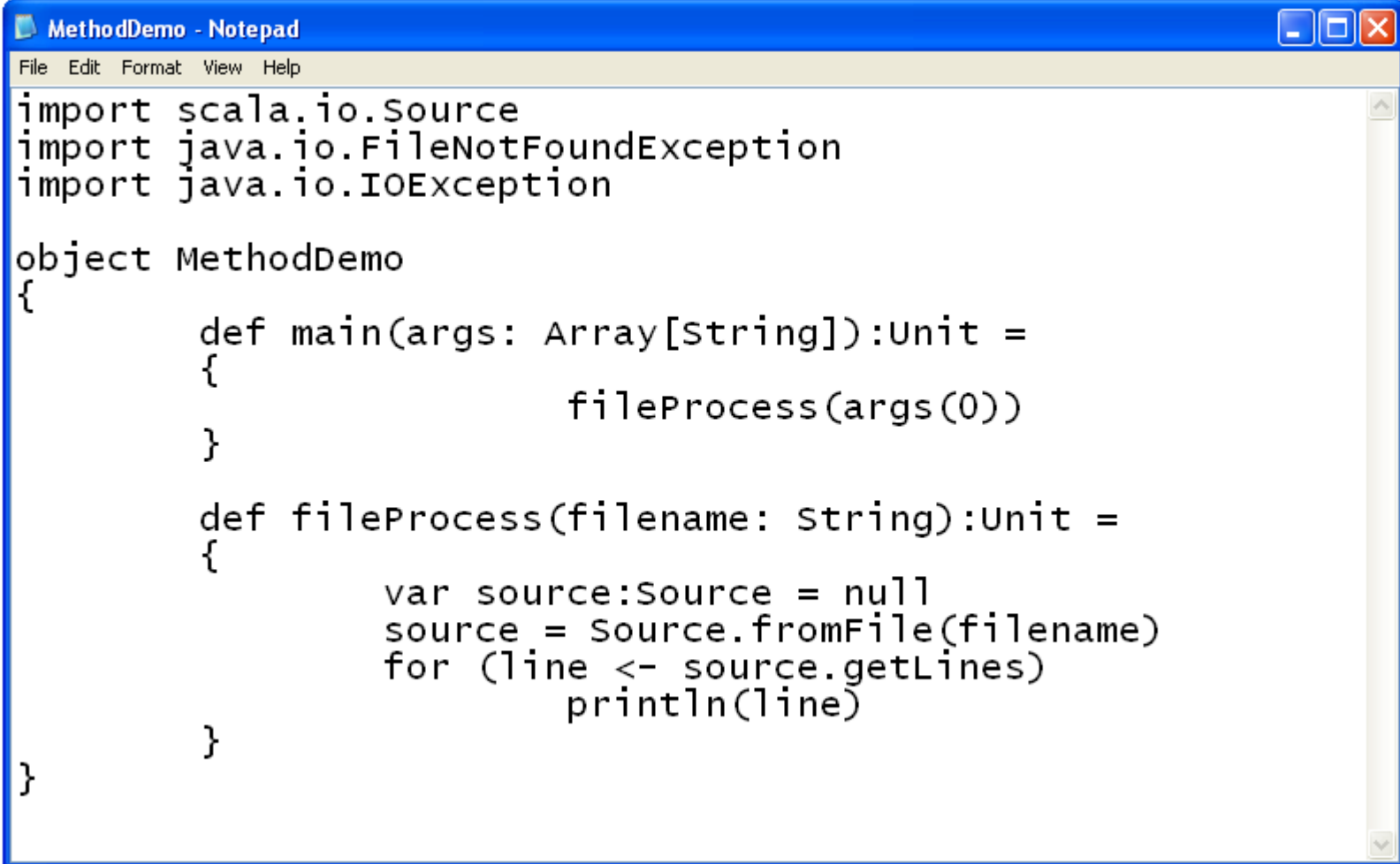
Introduction

- ❖ In addition to defining a function as a method, Scala allows us to define other types of functions such as local functions and anonymous ones.

Method

- ❖ The simplest most common form of a function is defining it as a method.
- ❖ Method is a simple function defined within the scope of a class or an object.

Method

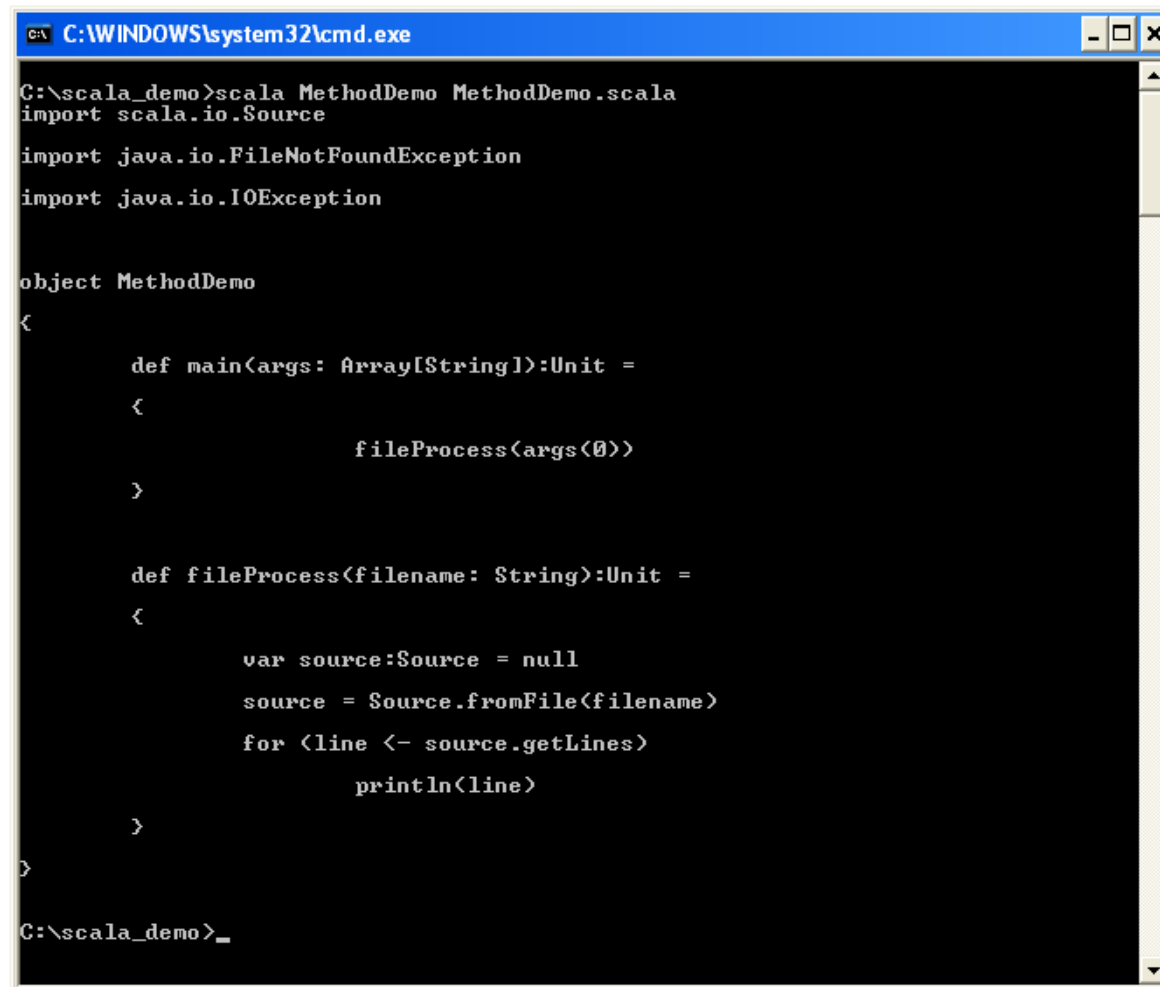


```
MethodDemo - Notepad
File Edit Format View Help
import scala.io.Source
import java.io.FileNotFoundException
import java.io.IOException

object MethodDemo
{
  def main(args: Array[String]):Unit =
  {
    fileProcess(args(0))
  }

  def fileProcess(filename: String):Unit =
  {
    var source:Source = null
    source = Source.fromFile(filename)
    for (line <- source.getLines)
      println(line)
  }
}
```

Method



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scala MethodDemo MethodDemo.scala
import scala.io.Source
import java.io.FileNotFoundException
import java.io.IOException

object MethodDemo
{
    def main(args: Array[String]):Unit =
    {
        fileProcess(args<0>)
    }

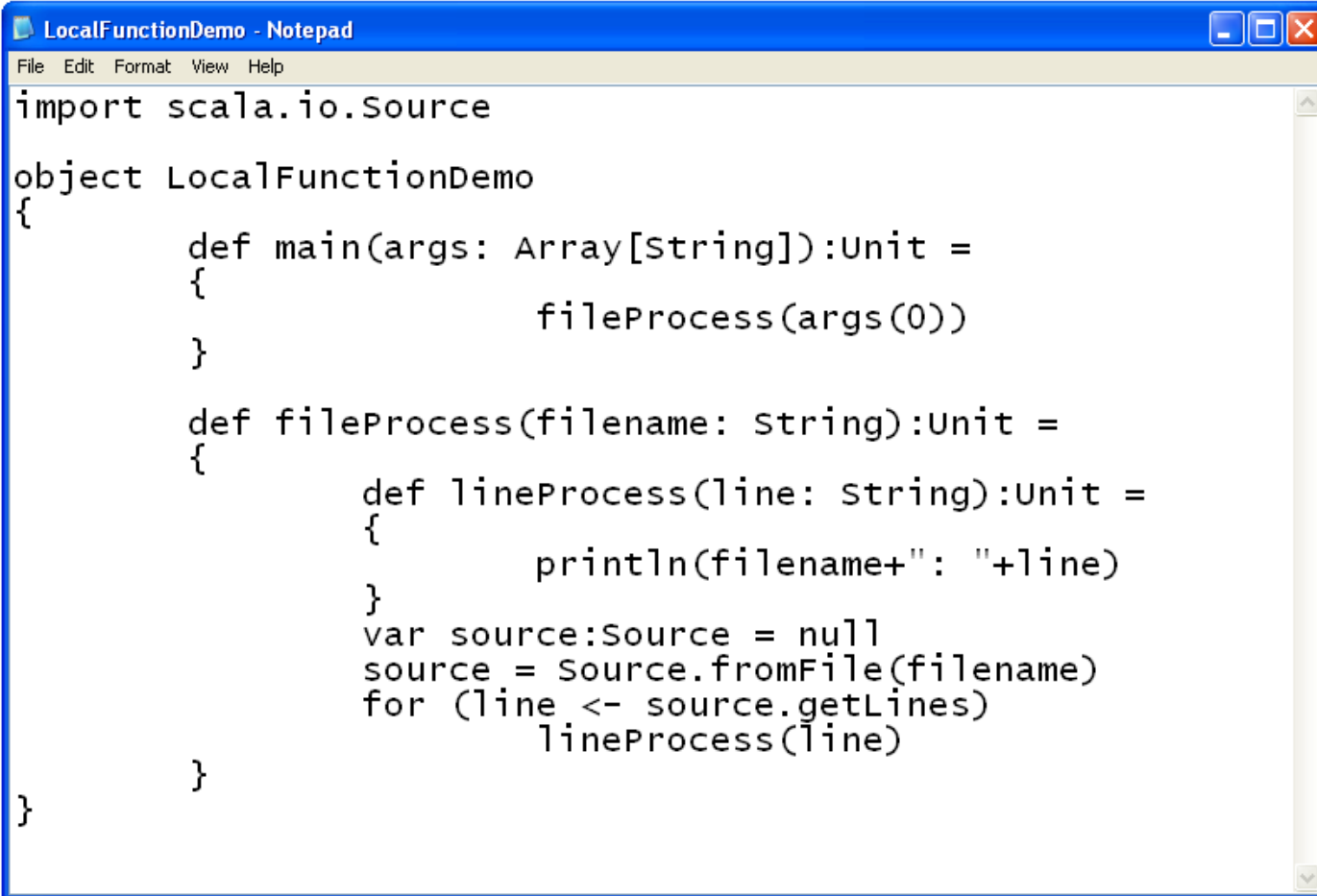
    def fileProcess(filename: String):Unit =
    {
        var source:Source = null
        source = Source.fromFile(filename)
        for (line <- source.getLines)
            println(line)
    }
}

C:\scala_demo>_
```

Local Functions

- ❖ The Scala programming language allows us to define a local function, which is a function we define within the scope of another function.
- ❖ Local functions are visible in their enclosing block only.
- ❖ The local function can use variables defined within the scope of its enclosing function.
- ❖ The local function can be invoked from within the scope of the outer function only.

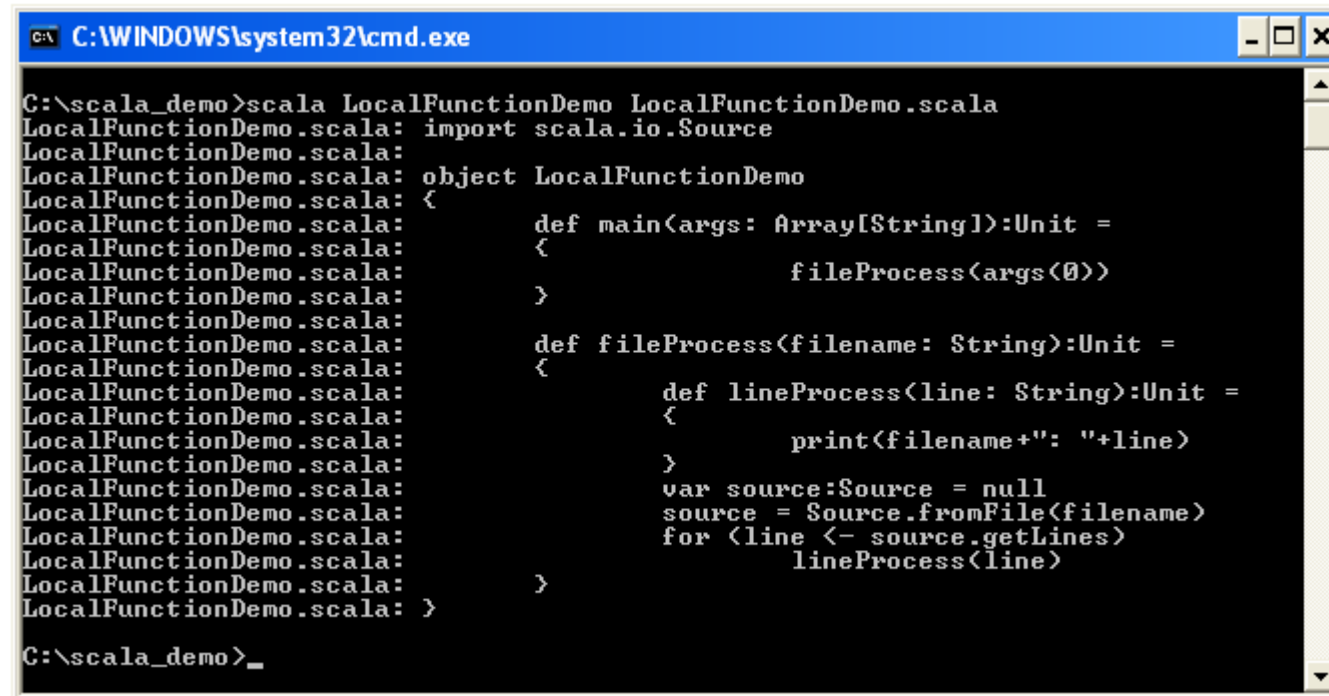
Local Functions



```
LocalFunctionDemo - Notepad
File Edit Format View Help
import scala.io.Source
object LocalFunctionDemo
{
  def main(args: Array[String]):Unit =
  {
    fileProcess(args(0))
  }

  def fileProcess(filename: String):Unit =
  {
    def lineProcess(line: String):Unit =
    {
      println(filename+": "+line)
    }
    var source:Source = null
    source = Source.fromFile(filename)
    for (line <- source.getLines)
      lineProcess(line)
  }
}
```

Local Functions



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scala LocalFunctionDemo LocalFunctionDemo.scala
LocalFunctionDemo.scala: import scala.io.Source
LocalFunctionDemo.scala:
LocalFunctionDemo.scala: object LocalFunctionDemo
LocalFunctionDemo.scala: {
LocalFunctionDemo.scala:     def main(args: Array[String]):Unit =
LocalFunctionDemo.scala:     {
LocalFunctionDemo.scala:         fileProcess(args(0))
LocalFunctionDemo.scala:     }
LocalFunctionDemo.scala:
LocalFunctionDemo.scala:     def fileProcess(filename: String):Unit =
LocalFunctionDemo.scala:     {
LocalFunctionDemo.scala:         def lineProcess(line: String):Unit =
LocalFunctionDemo.scala:         {
LocalFunctionDemo.scala:             print(filename+": "+line)
LocalFunctionDemo.scala:         }
LocalFunctionDemo.scala:         var source:Source = null
LocalFunctionDemo.scala:         source = Source.fromFile(filename)
LocalFunctionDemo.scala:         for (line <- source.getLines)
LocalFunctionDemo.scala:             lineProcess(line)
LocalFunctionDemo.scala:     }
LocalFunctionDemo.scala: }
C:\scala_demo>_
```


Anonymous Functions

- ❖ We can define a function without a name, and pass it over whether into a specific variable or into a method parameter as if it was a value.
- ❖ During run-time each anonymous function is compiled into a class and instantiated. The instantiated object is the value we pass over.

Anonymous Functions

- ❖ The anonymous functions are also known as first class functions.

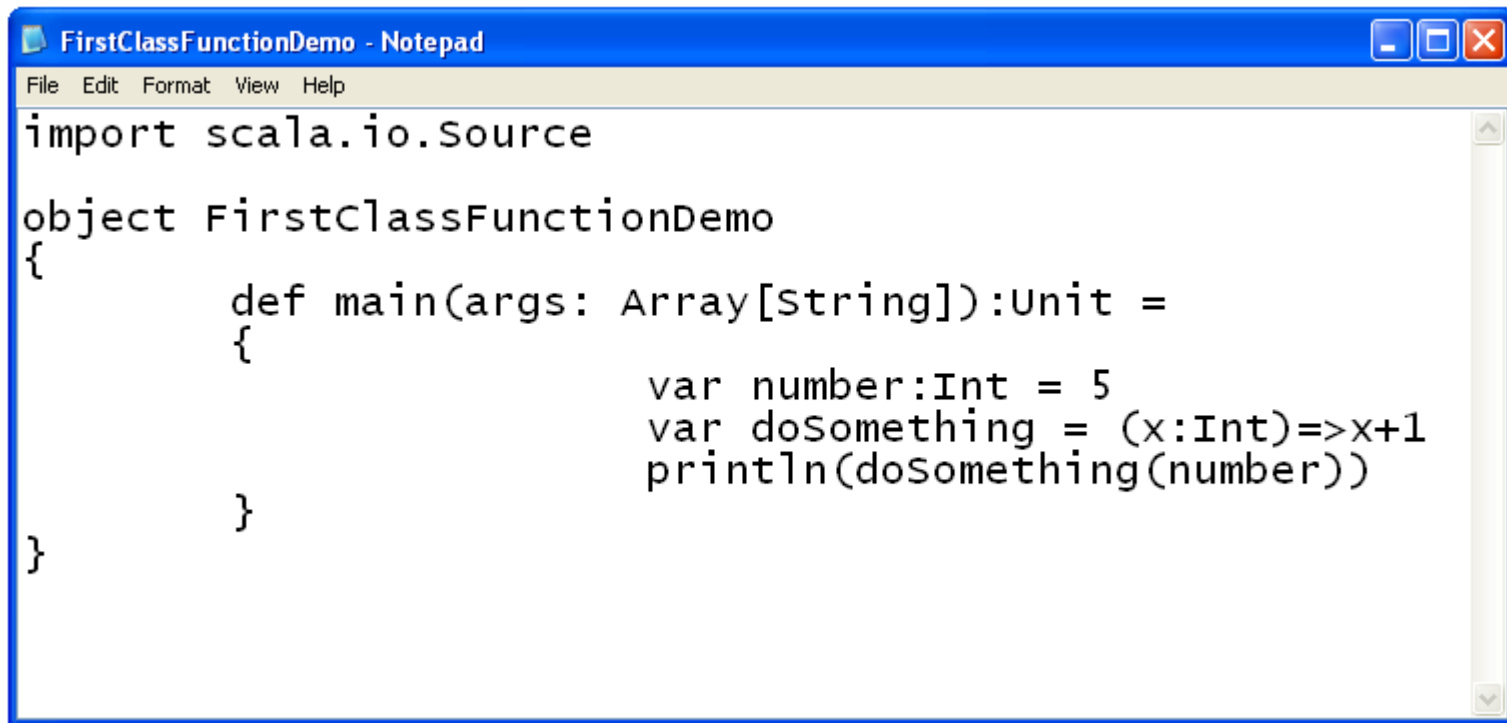
...

```
var increment = (i:Int) => i+1
```

```
var num = increment(5)
```

...

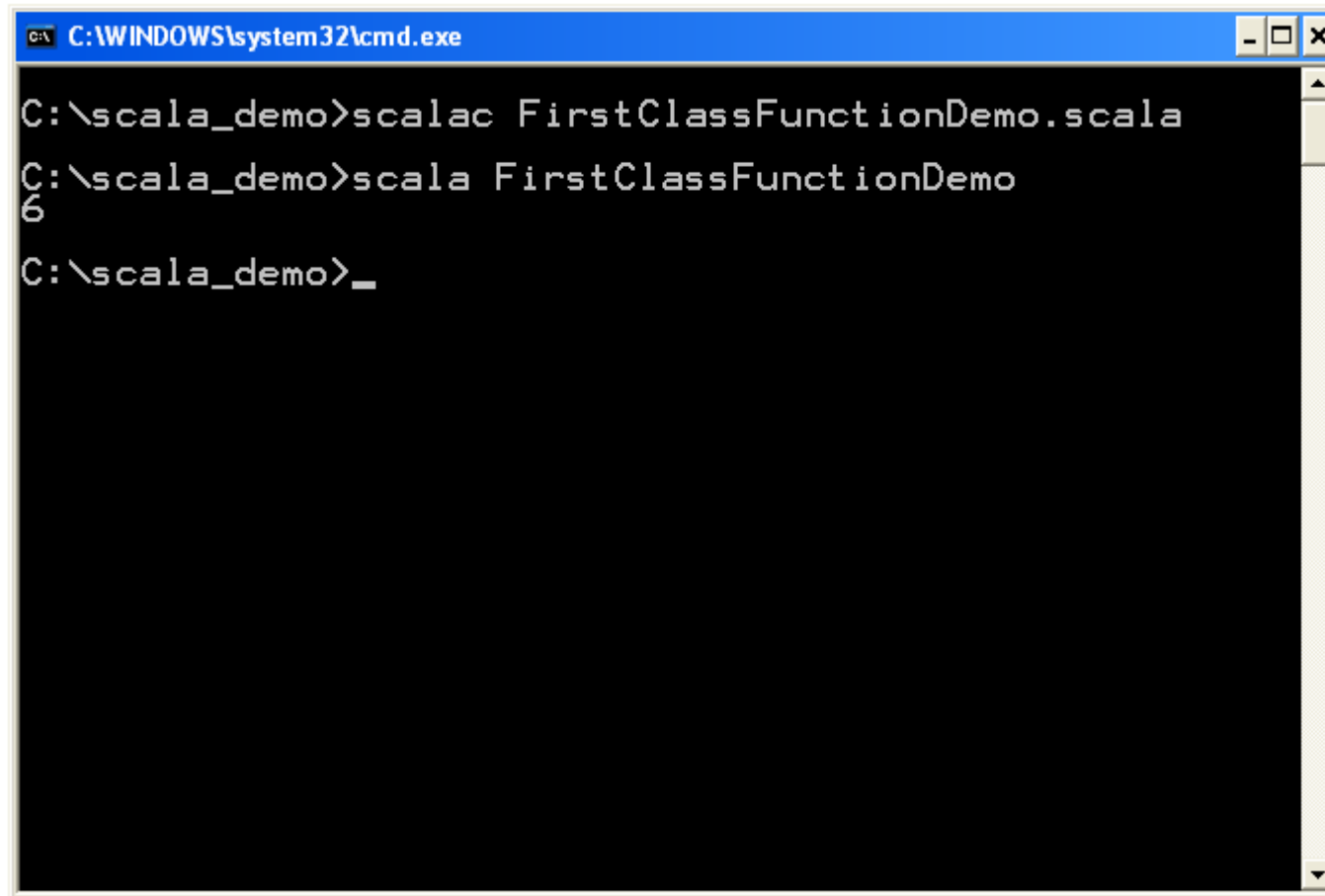
Anonymous Functions



```
File Edit Format View Help
import scala.io.Source

object FirstClassFunctionDemo
{
    def main(args: Array[String]):Unit =
    {
        var number:Int = 5
        var doSomething = (x:Int)=>x+1
        println(doSomething(number))
    }
}
```

Anonymous Functions

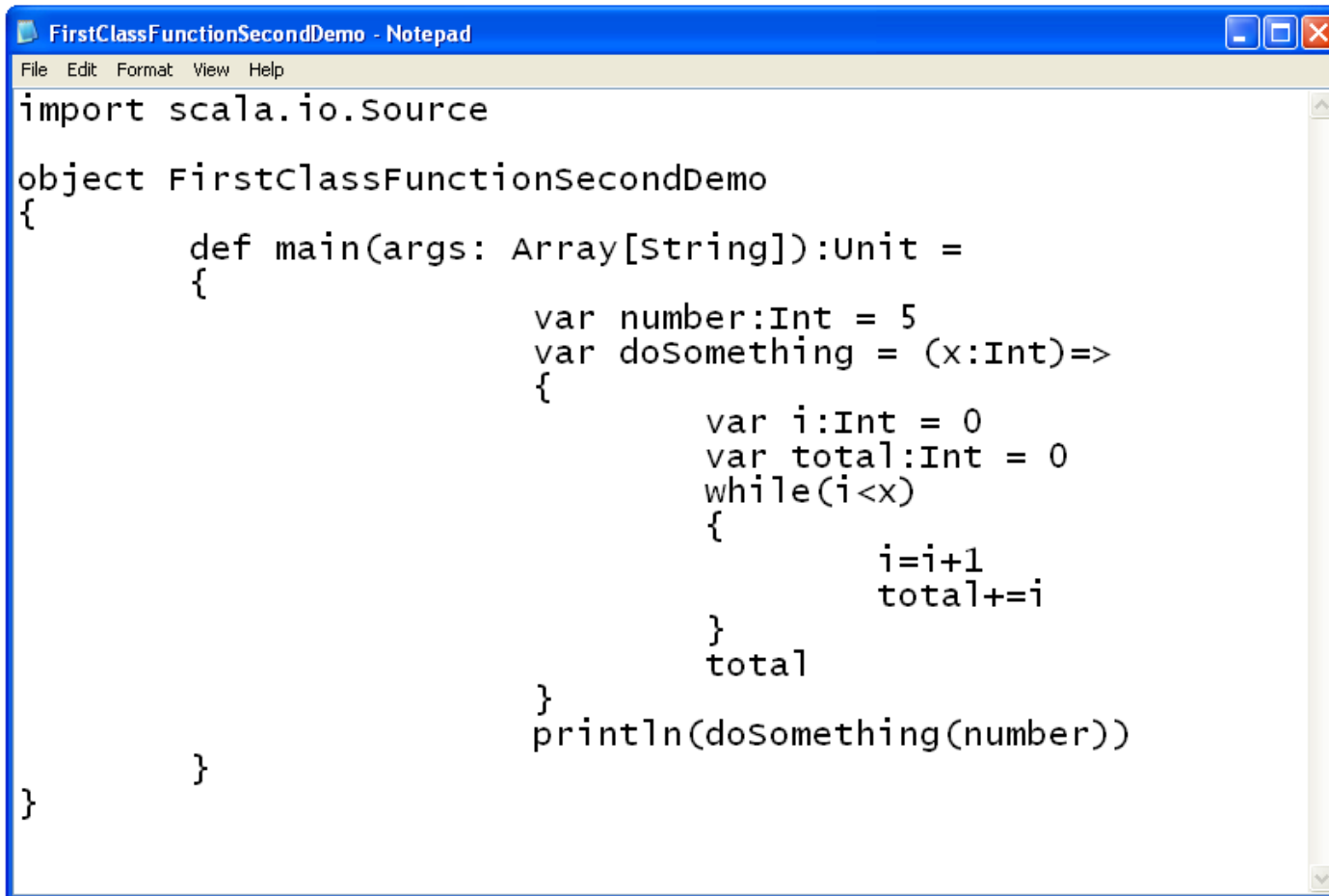


```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac FirstClassFunctionDemo.scala
C:\scala_demo>scala FirstClassFunctionDemo
6
C:\scala_demo>_
```

Anonymous Functions

- ❖ When defining an anonymous function we can place more than one statement. We should place all statements within a block.

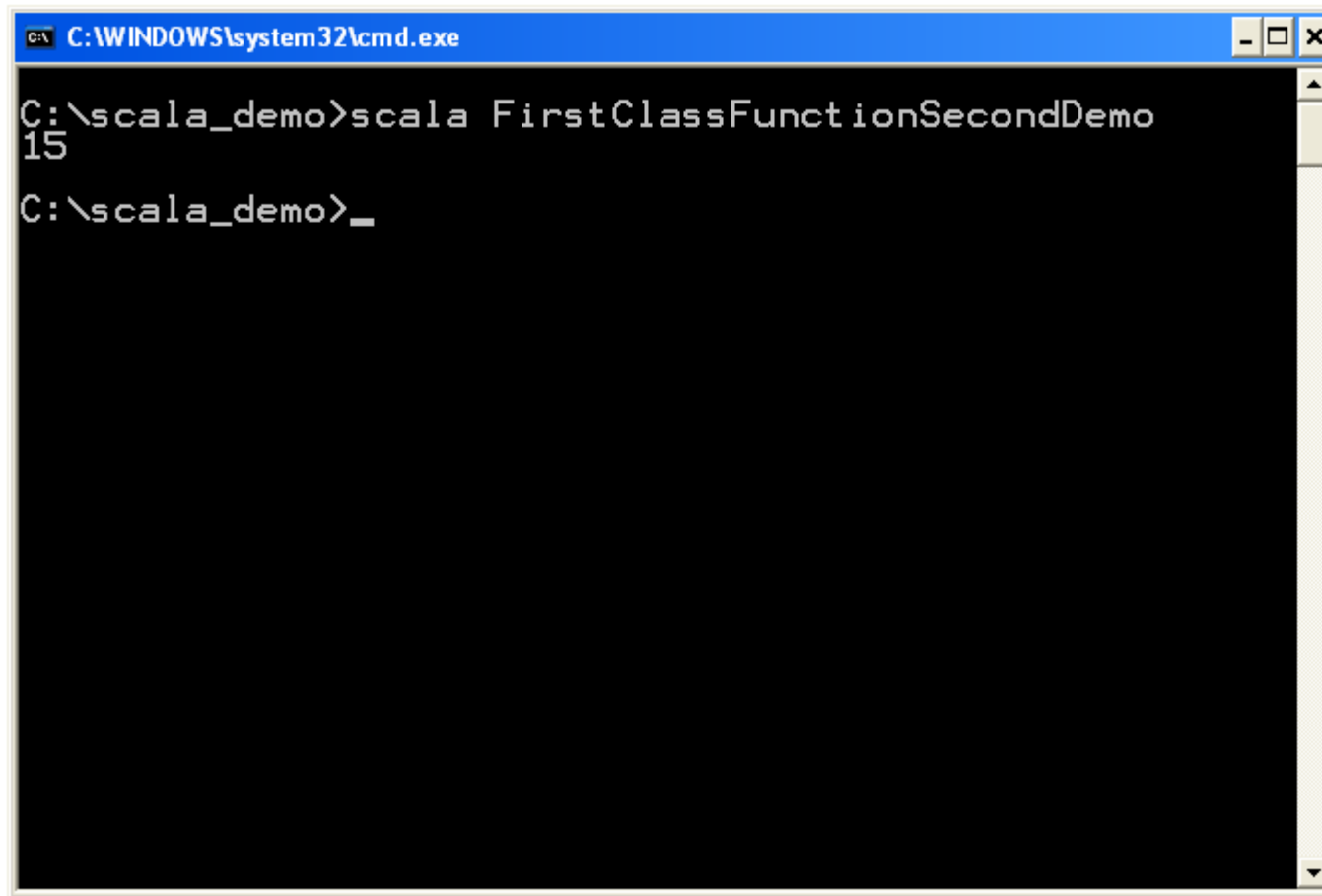
Anonymous Functions



```
FirstClassFunctionSecondDemo - Notepad
File Edit Format View Help
import scala.io.Source

object FirstClassFunctionSecondDemo
{
  def main(args: Array[String]):Unit =
  {
    var number:Int = 5
    var doSomething = (x:Int)=>
    {
      var i:Int = 0
      var total:Int = 0
      while(i<x)
      {
        i=i+1
        total+=i
      }
      total
    }
    println(doSomething(number))
  }
}
```

Anonymous Functions

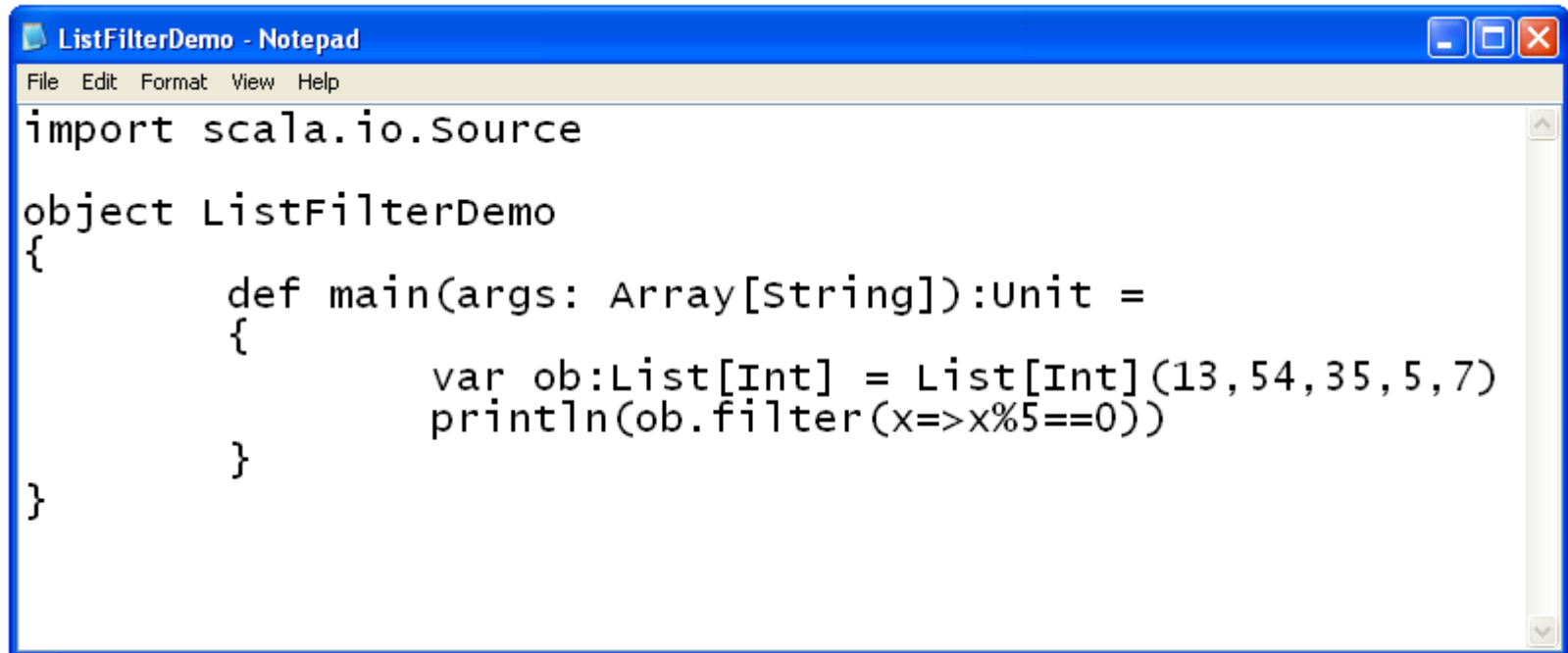


```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scala FirstClassFunctionSecondDemo
15
C:\scala_demo>_
```

Anonymous Functions

- ❖ Many of the classes the Scala library includes already allow us to use functions literals passing them over to functions we call.
- ❖ One example is the filter method we can call on a List object passing over a function that once called on each one of the List elements it returns true or false.
- ❖ It is possible to omit the parameters' types.

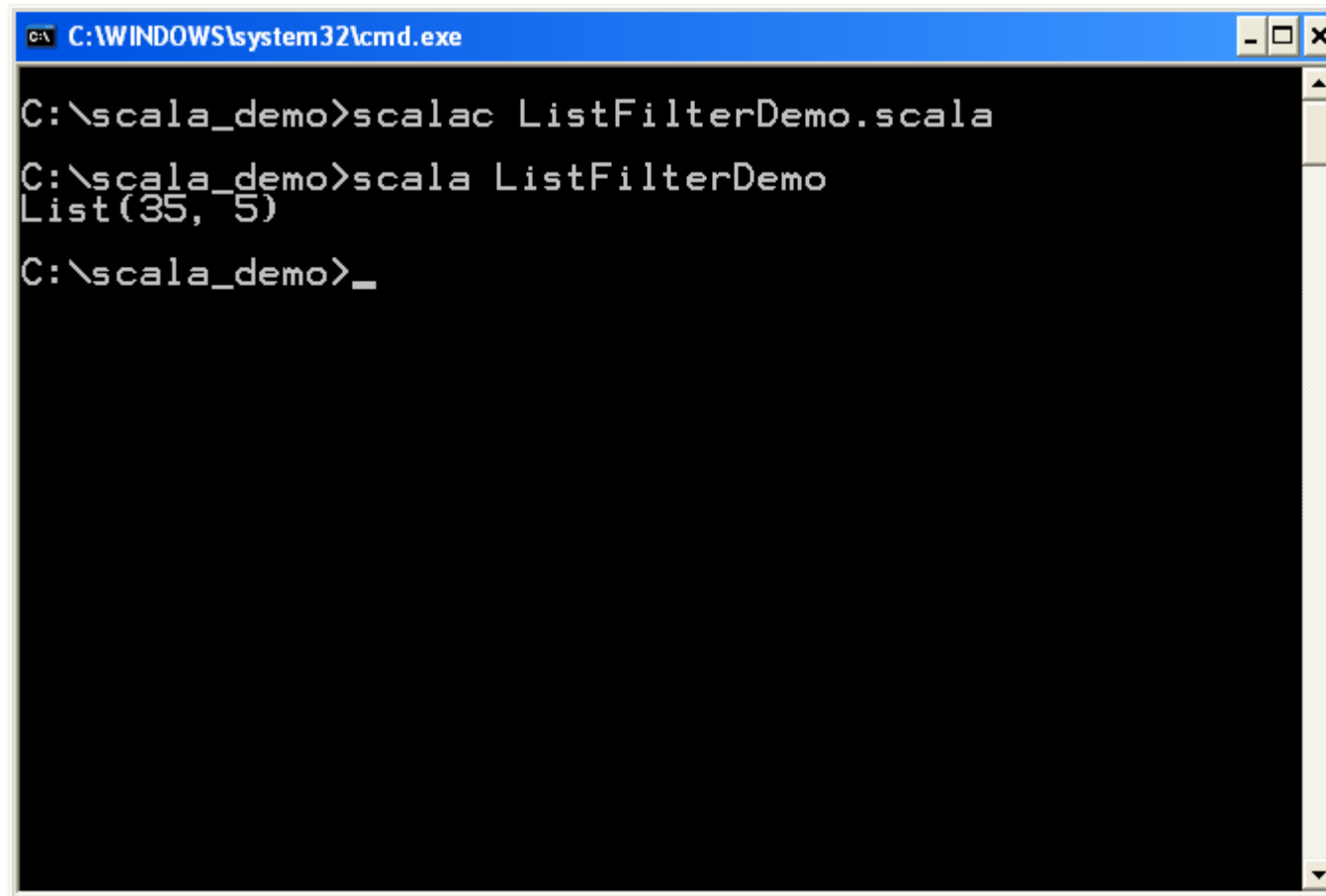
Anonymous Functions



```
ListFilterDemo - Notepad
File Edit Format View Help
import scala.io.Source

object ListFilterDemo
{
    def main(args: Array[String]):Unit =
    {
        var ob:List[Int] = List[Int](13,54,35,5,7)
        println(ob.filter(x=>x%5==0))
    }
}
```

Anonymous Functions



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac ListFilterDemo.scala
C:\scala_demo>scala ListFilterDemo
List(35, 5)
C:\scala_demo>_
```

Anonymous Functions

- ❖ When creating an anonymous function and passing it over as an argument to another function we can use a placeholder and avoid the left part of the function.

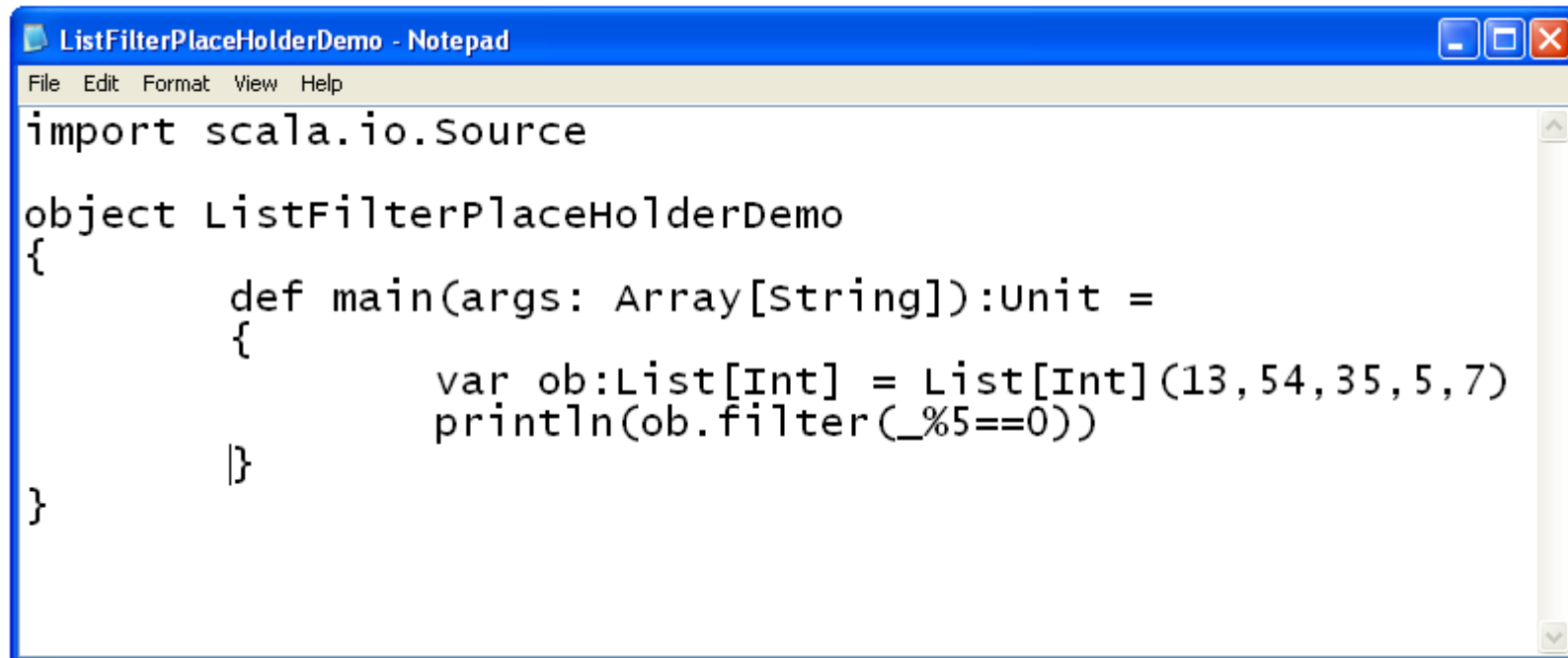
...

```
var ob:List[Int] = List[Int](13,54,35,5,7)
```

```
println(ob.filter(_%5==0))
```

...

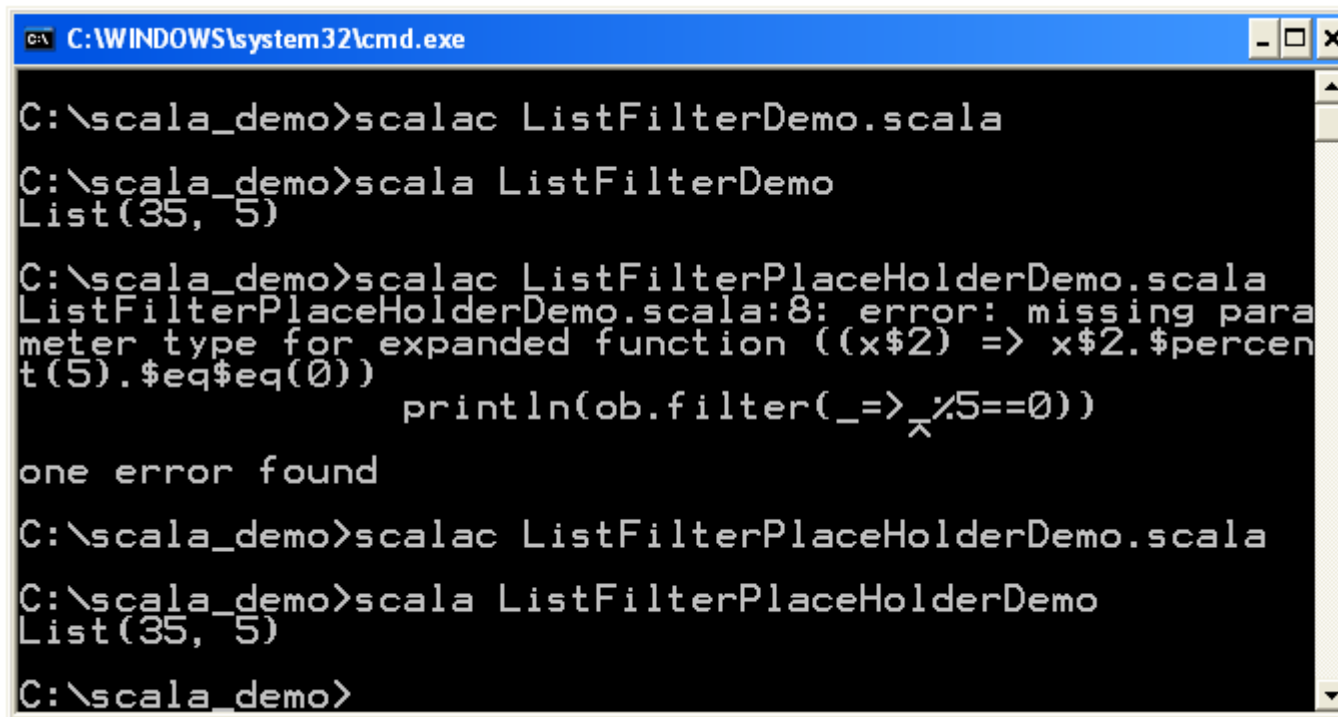
Anonymous Functions



```
ListFilterPlaceholderDemo - Notepad
File Edit Format View Help
import scala.io.Source

object ListFilterPlaceholderDemo
{
    def main(args: Array[String]):Unit =
    {
        var ob:List[Int] = List[Int](13,54,35,5,7)
        println(ob.filter(_%5==0))
    }
}
```

Anonymous Functions



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac ListFilterDemo.scala
C:\scala_demo>scala ListFilterDemo
List(35, 5)
C:\scala_demo>scalac ListFilterPlaceholderDemo.scala
ListFilterPlaceholderDemo.scala:8: error: missing parameter
type for expanded function ((x$2) => x$2.$percent(5).$eq$eq(0))
    println(ob.filter(_=>_%5==0))
                        ^
one error found
C:\scala_demo>scalac ListFilterPlaceholderDemo.scala
C:\scala_demo>scala ListFilterPlaceholderDemo
List(35, 5)
C:\scala_demo>
```

Anonymous Functions

- ❖ When using underscores as placeholders for parameters the compiler might not have enough information in order to infer the missing parameter types. In these cases we can specify the type using a colon in the following way.

...

```
var doSomething = (_:Double)+(_:Double)
```

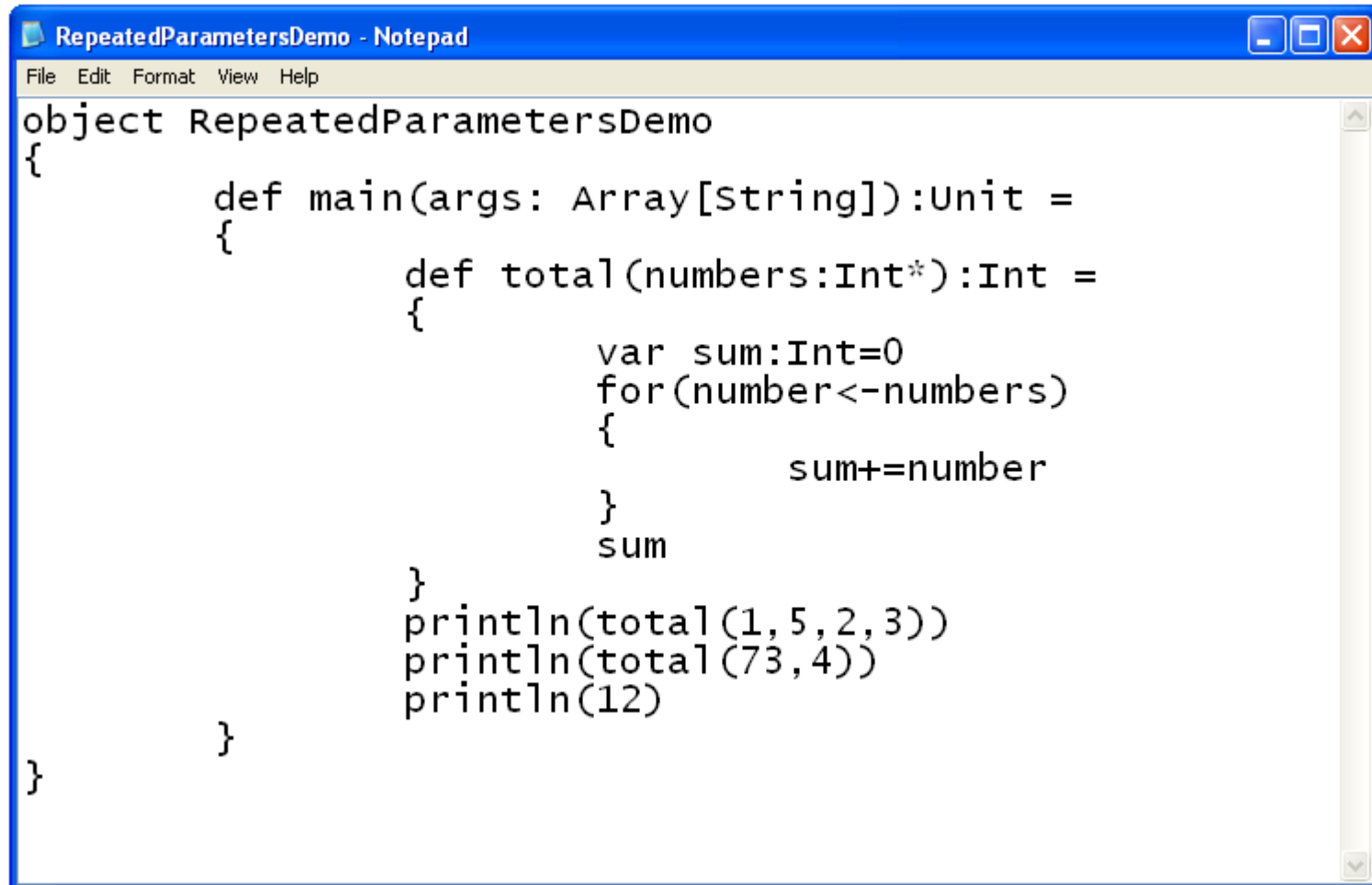
```
var num = doSomething(5,4.2)
```

...

Repeated Parameters

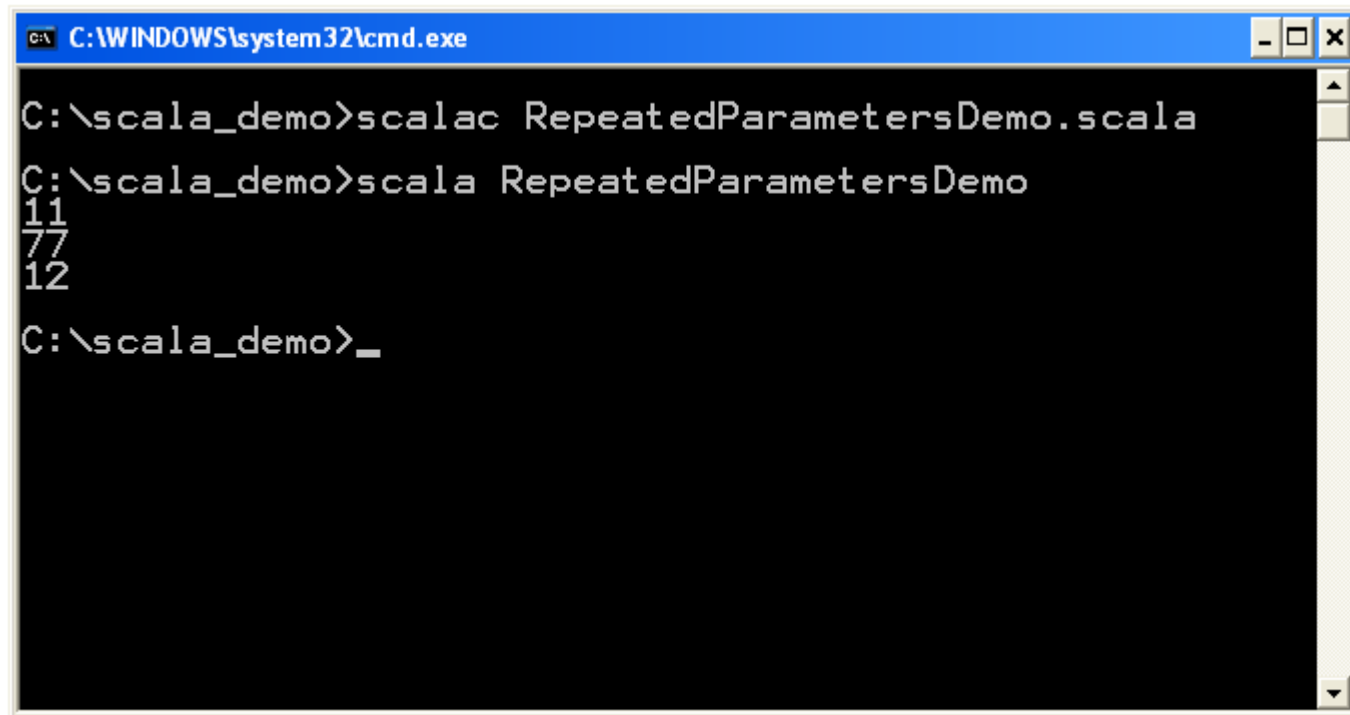
- ❖ When placing an asterisk after the type of the last parameter we allow calling our function with a variable number of arguments. The last argument can be passed over any number of times. Including 0.

Repeated Parameters



```
object RepeatedParametersDemo
{
  def main(args: Array[String]):Unit =
  {
    def total(numbers:Int*):Int =
    {
      var sum:Int=0
      for(number<-numbers)
      {
        sum+=number
      }
      sum
    }
    println(total(1,5,2,3))
    println(total(73,4))
    println(12)
  }
}
```


Repeated Parameters



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac RepeatedParametersDemo.scala
C:\scala_demo>scala RepeatedParametersDemo
11
77
12
C:\scala_demo>_
```

Tail Recursion

- ❖ If the last action a function performs is calling to itself then it is a tail recursive function. When a tail recursive function is executed the computer doesn't need to keep the memory stack frames. It can use one frame only.
- ❖ When having tail recursion we can use the `@tailrec` in order to instruct the compiler to avoid keeping the stack frames and use one frame only. Doing so the performance will be significantly improved.

Tail Recursion

```
import annotation.tailrec

object Program
{
  def main(args: Array[String]):Unit =
  {
    println(factorial(4))
  }

  def factorial(num:Int):Int =
  {
    @tailrec
    def calculate(accumulator:Int,number:Int):Int =
    {
      if(number==0)
        accumulator
      else
        calculate(accumulator*number,number-1)
    }
    calculate(1,num)
  }
}
```



Function Type

- ❖ We can define variables, function parameters and even function returned values to be of a function type.

$(A, B) \Rightarrow C$

The A,B and C letters stand for types.

Function Type

```
import annotation.tailrec

object Program
{
  def main(args: Array[String]):Unit =
  {
    var func:(Int,Int)=>Int = sum;
    println(func(4,3))
    func = multiply
    println(func(4,3))
  }
  def sum(a:Int,b:Int):Int = a+b
  def multiply(a:Int,b:Int):Int = a*b
}
```



By Names Parameters

- ❖ When calling a function and passing over an argument which is an expression that needs to be evaluated, the expression will be evaluated before the function is invoked and its value will be passed over. This is the default behavior.
- ❖ By adding `=>` in between the parameter name and its type we will defer the expression evaluation into the function execution to be performed when its value is required.

By Names Parameters

```
package il.ac.hit.samples

object Program
{
  def main(args: Array[String])
  {
    println(System.currentTimeMillis())
    printWithDelay(System.currentTimeMillis())
  }

  def printWithDelay( t: => Long) =
  {
    Thread.sleep(10000)
    println(t)
  }
}
```



Function Values are Objects

- ❖ Function values are treated as objects. The function $A \Rightarrow B$ is an abbreviation for using a new object instantiated from a class that extends the `scala.Function1[A, B]` trait and overrides the `apply` function.
- ❖ There are currently `Function1`, `Function2`, `Function3`... etc... up to `Function22`, that takes 22 parameters.

Function Values are Objects

```
object HelloSample
{
  def main(args:Array[String]):Unit =
  {
    val func1 = (num:Int) => 2*num
    println(func1(4))
    val func2 = new MyFunction
    println(func2(4))
  }
}

// (num:Int) => 2*num
class MyFunction extends Function1[Int,Int]
{
  def apply(num:Int) = 2*num
}
```



Anonymous Class Syntax

- ❖ When calling a function we indirectly invoke the apply method on the object that represents the function. We can use the anonymous class syntax.

```
val func = (x:Int) => 2 * x  
func(3)
```

would be equivalent to:

```
val func = new Function1[Int,Int]  
{  
    def apply(x:Int) = 2 * x  
}  
func.apply(3)
```



The Postfix Notation

- ❖ When enabling the postfix notation we can call a method on a specific object without using the dot (.).
- ❖ In order to enable this unique feature we should add the following to our code

```
import scala.language.postfixOps
```

The Postfix Notation

```
package il.ac.hit.courses.functional.scala.samples

import scala.language.postfixOps

object PostfixDemo {
  def main(args: Array[String]): Unit = {
    var ob = new Something()
    var text = ob toString
  }
}

class Something
```

The Curley Brackets Syntax

- ❖ When calling a function we can use curly brackets instead of parentheses for passing over the arguments to the function.

```
object TheBracketsSyntax {  
  def main(args: Array[String]): Unit = {  
    //gaga(5)  
    gaga {5}  
  }  
  def gaga(num: Int):Unit = {  
    print(num)  
  }  
}
```

Symbolic Methods

- ❖ Scala allows us to define methods their names includes symbols instead of regular letters. Given the infix optional notation we can use that for creating new customized operators.

Symbolic Methods

```
object Program {
  def main(args: Array[String]): Unit = {
    var a = new Circle(4)
    var b = new Circle(5)
    var c = a ^+^ b
  }
}

class Circle(rad: Double)
{
  private var radius = rad
  def ^+^ (ob:Circle):Circle = new Circle(this.radius+ob.radius)
  override def toString():String = "radius="+radius
}
```