

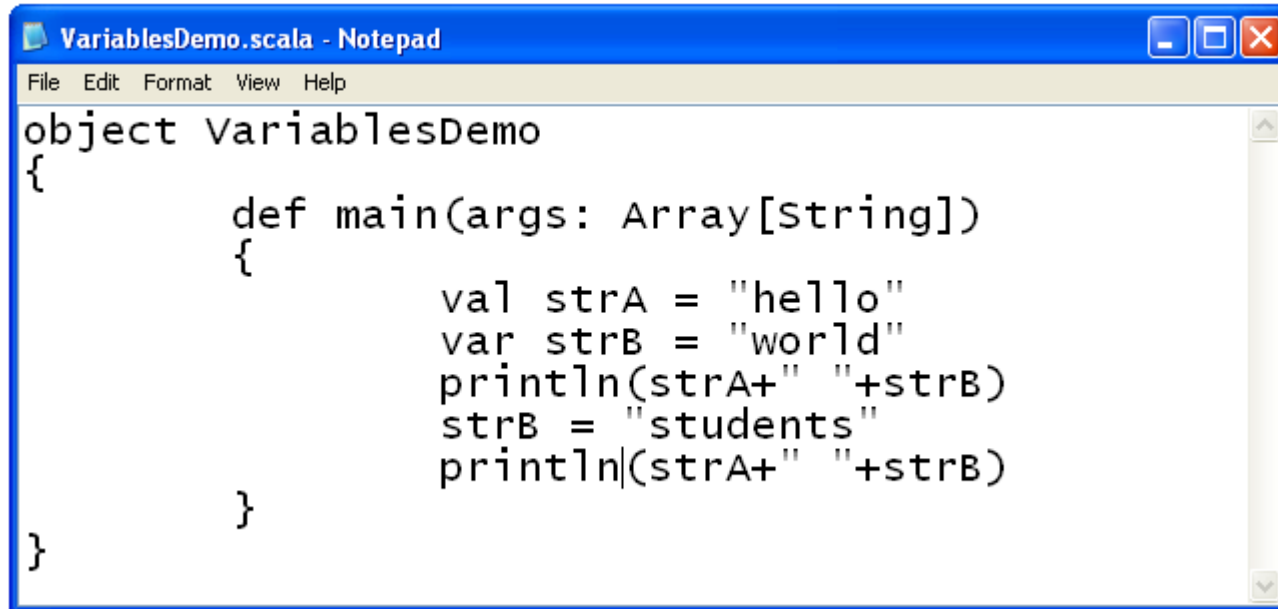
# Basics

# Define Variables

- ❖ Scala supports two kinds of variables: `vals` and `vars`. A `val` is similar to a final variable in Java. Once a `val` variable was initialize it can never be reassigned. A `var` is similar to a non-final variable in Java. We can reassign it with a new value.

```
...  
val strA = "hello"  
...  
var strB = "world"  
...
```

# Define Variables



```
object VariablesDemo
{
    def main(args: Array[String])
    {
        val strA = "hello"
        var strB = "world"
        println(strA+" "+strB)
        strB = "students"
        println(strA+" "+strB)
    }
}
```

# Variables Type Inference

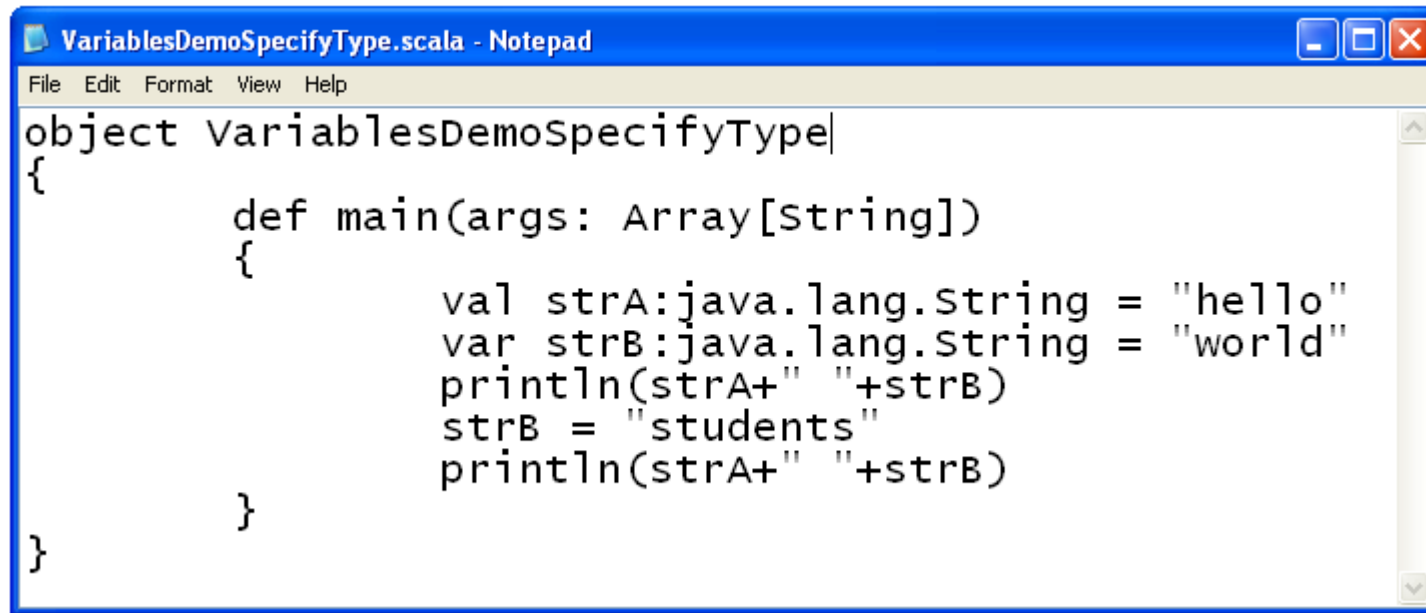
- ❖ When defining a variable, the Scala compiler has the ability to figure out the exact type in accordance with the value we assign.

# Specifying Variable Type

- ❖ We can specify the exact type. Doing so might improve the readability of our code and improves its documentation.

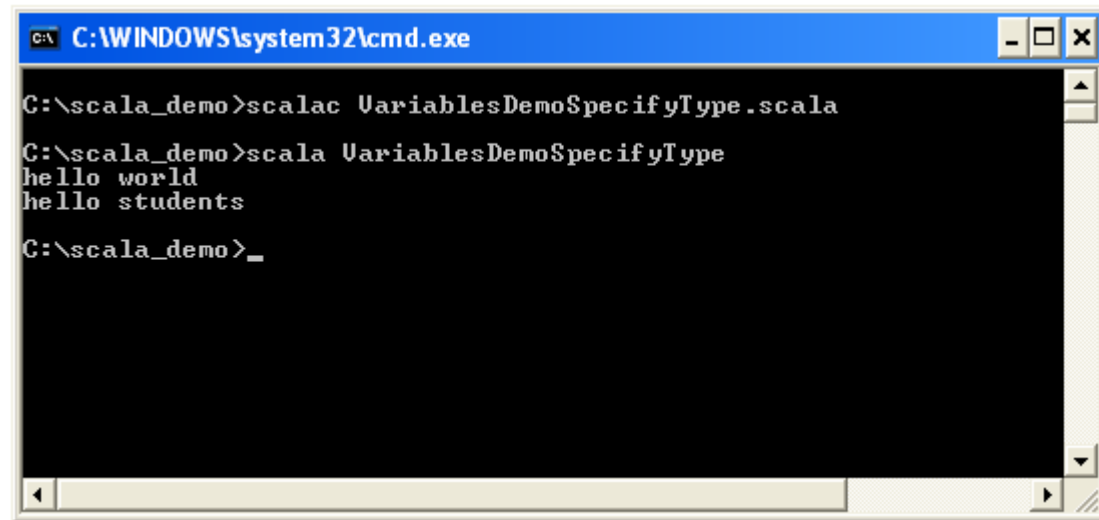
```
...  
val strA:java.lang.String = "hello"  
...  
var strB:java.lang.String = "world"  
...
```

# Specifying Variable Type

A screenshot of a Notepad window titled "VariablesDemoSpecifyType.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The code inside is as follows:

```
object VariablesDemoSpecifyType
{
    def main(args: Array[String])
    {
        val strA:java.lang.String = "hello"
        var strB:java.lang.String = "world"
        println(strA+" "+strB)
        strB = "students"
        println(strA+" "+strB)
    }
}
```

# Specifying Variable Type



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac VariablesDemoSpecifyType.scala

C:\scala_demo>scala VariablesDemoSpecifyType
hello world
hello students

C:\scala_demo>_
```

# Define Functions

- ❖ Defining a function is done using the 'def' keyword followed with the function name followed with a comma separated list of parameters in parentheses.
- ❖ Each parameter must be specified together with its type.
- ❖ After the close parenthesis we will specify the type of the returned value together with the '=' character.



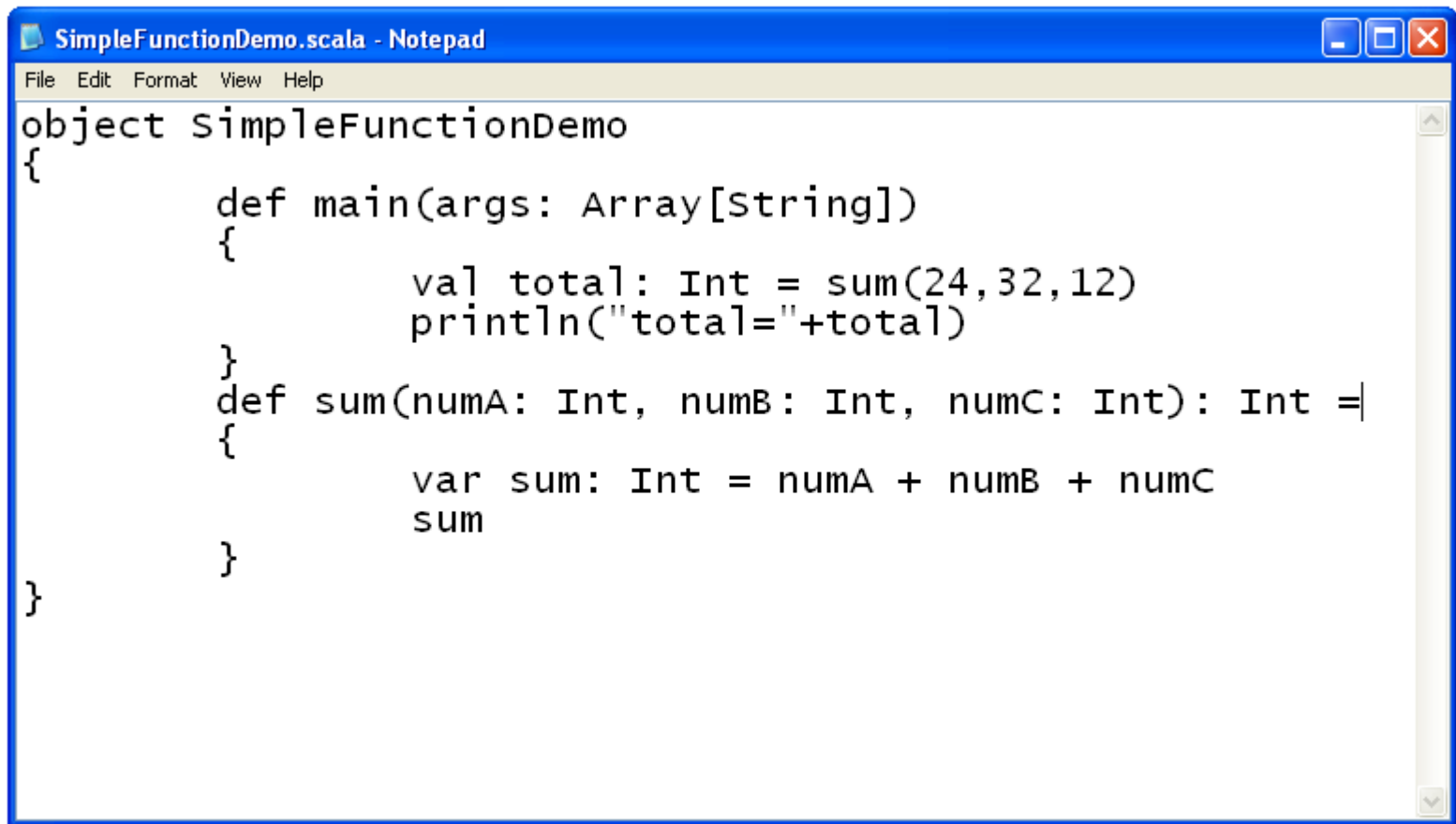
# Define Functions

- ❖ We must specify the types of the parameters. We cannot avoid that.

```
def sum(numA: Int, numB: Int, numC: Int): Int =  
{  
    var sum: Int = numA + numB + numC  
    sum  
}
```

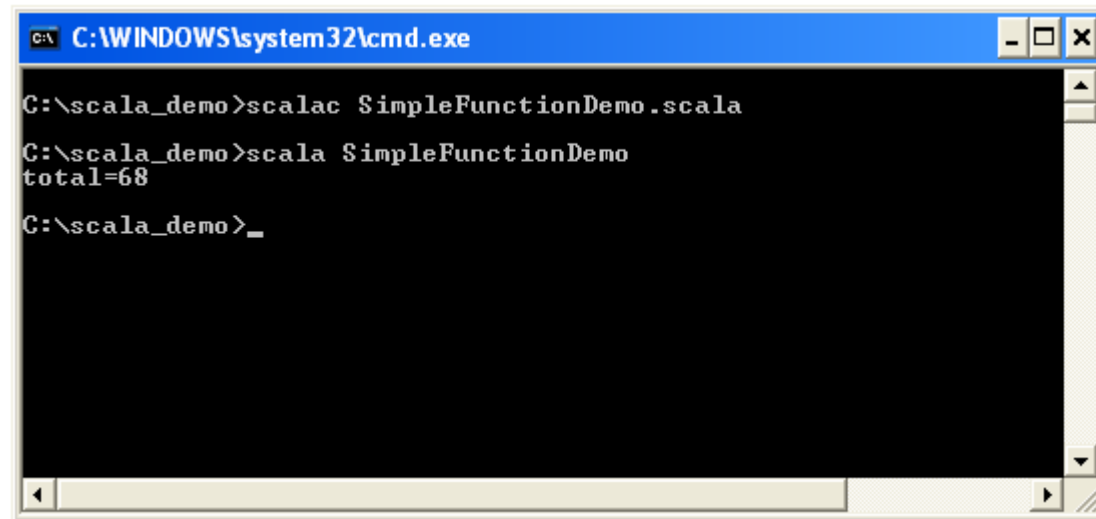
- ❖ We don't use the `return` keyword. The `=` character following the function title cannot be avoided. It reflects the fact that a function defines an expression that has a specific value.

# Define Functions



```
SimpleFunctionDemo.scala - Notepad
File Edit Format View Help
object SimpleFunctionDemo
{
    def main(args: Array[String])
    {
        val total: Int = sum(24,32,12)
        println("total="+total)
    }
    def sum(numA: Int, numB: Int, numC: Int): Int =
    {
        var sum: Int = numA + numB + numC
        sum
    }
}
```

# Define Functions



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac SimpleFunctionDemo.scala
C:\scala_demo>scala SimpleFunctionDemo
total=68
C:\scala_demo>_
```

# Define Functions

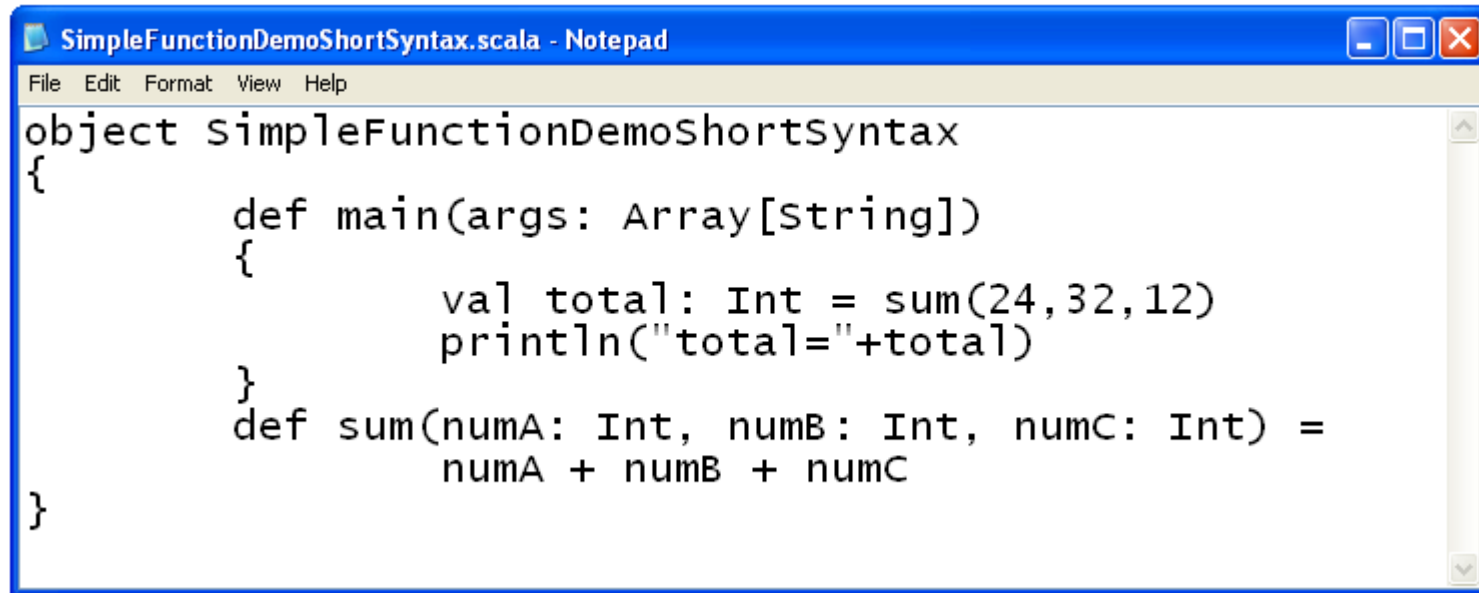
- ❖ When the function consists of one line only we can omit the curly braces.

```
def sum(numA: Int, numB: Int, numC: Int): Int = numA + numB + numC
```

- ❖ In many cases we can also omit the type of the returned value.

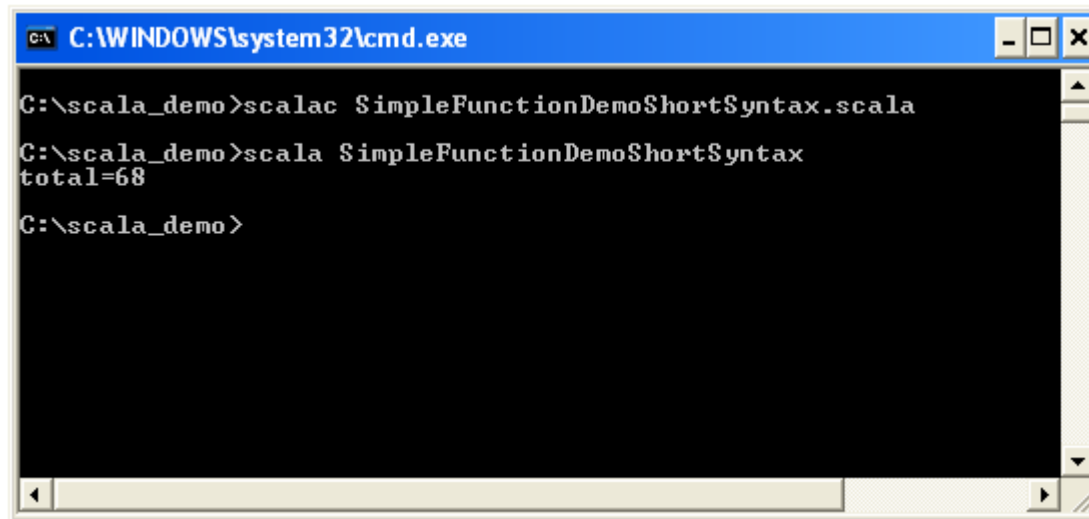
```
def sum(numA: Int, numB: Int, numC: Int) = numA + numB + numC
```

# Define Functions

A screenshot of a Notepad window titled "SimpleFunctionDemoShortSyntax.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The text area contains the following Scala code:

```
object SimpleFunctionDemoShortSyntax
{
    def main(args: Array[String])
    {
        val total: Int = sum(24,32,12)
        println("total="+total)
    }
    def sum(numA: Int, numB: Int, numC: Int) =
        numA + numB + numC
}
```

# Define Functions

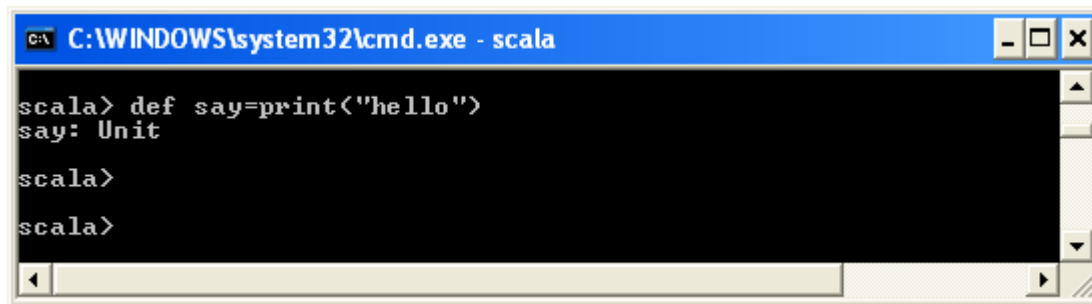


```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac SimpleFunctionDemoShortSyntax.scala
C:\scala_demo>scala SimpleFunctionDemoShortSyntax
total=68
C:\scala_demo>
```

# The Unit Type

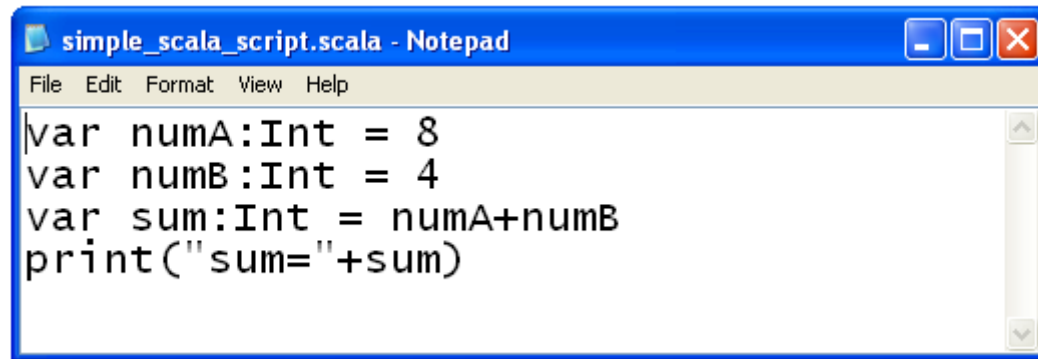
- ❖ When defining a function that doesn't have a returned value the type of its returned value would be `Unit`, Scala's equivalent for `void`.



```
C:\WINDOWS\system32\cmd.exe - scala
scala> def say=print("hello")
say: Unit
scala>
scala>
```

# Writing Scripts

- ❖ When writing code in Scala we can easily execute it as if it was a script file. We should execute scala.exe utility passing over the name of our file.

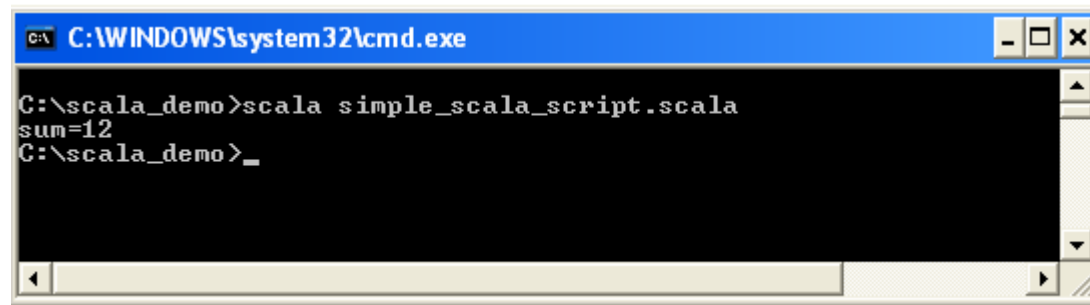


```
File Edit Format View Help
var numA:Int = 8
var numB:Int = 4
var sum:Int = numA+numB
print("sum="+sum)
```



# Writing Scripts

- ❖ We will execute this file by running the scala.exe utility.

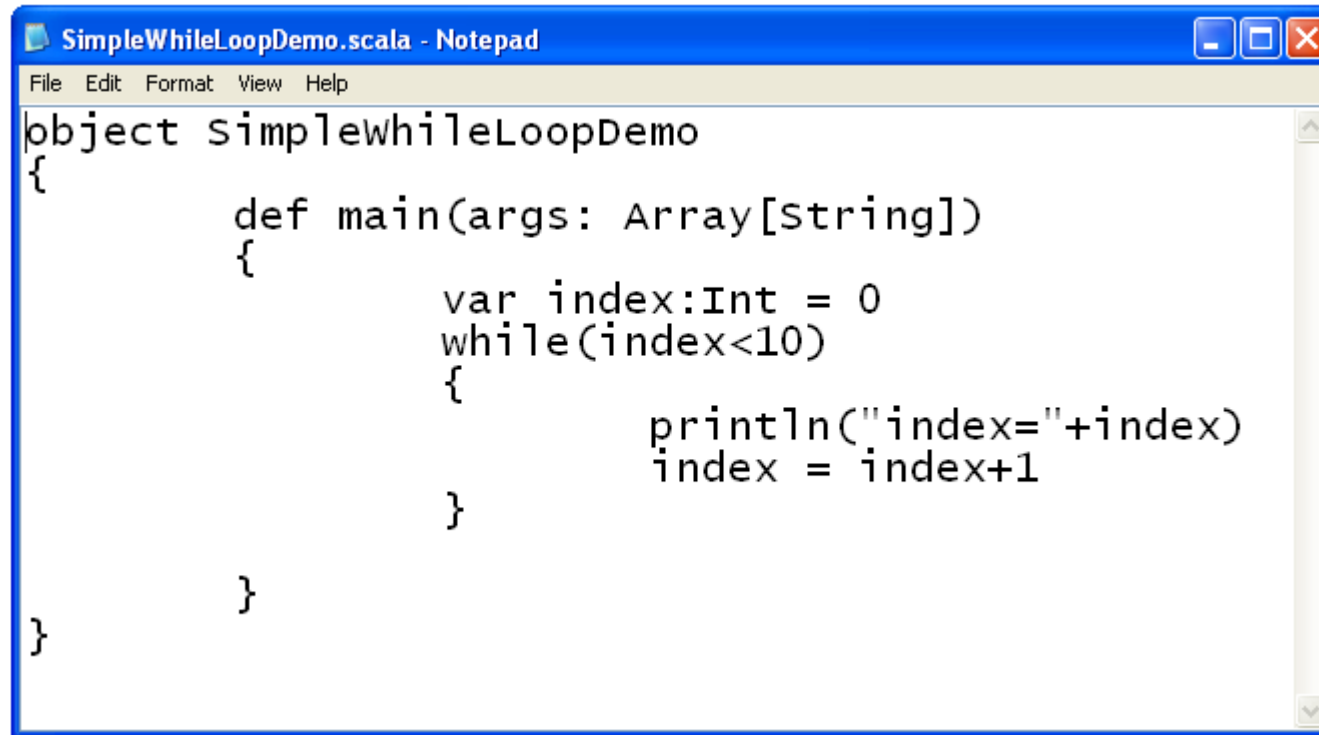


```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scala simple_scala_script.scala
sum=12
C:\scala_demo>_
```

The image shows a screenshot of a Windows command prompt window. The title bar at the top reads "C:\WINDOWS\system32\cmd.exe". The command prompt shows the following sequence of text: "C:\scala\_demo>scala simple\_scala\_script.scala", followed by the output "sum=12", and then the prompt "C:\scala\_demo>\_" with a cursor. The window has a standard Windows XP-style interface with a blue title bar and scrollbars.

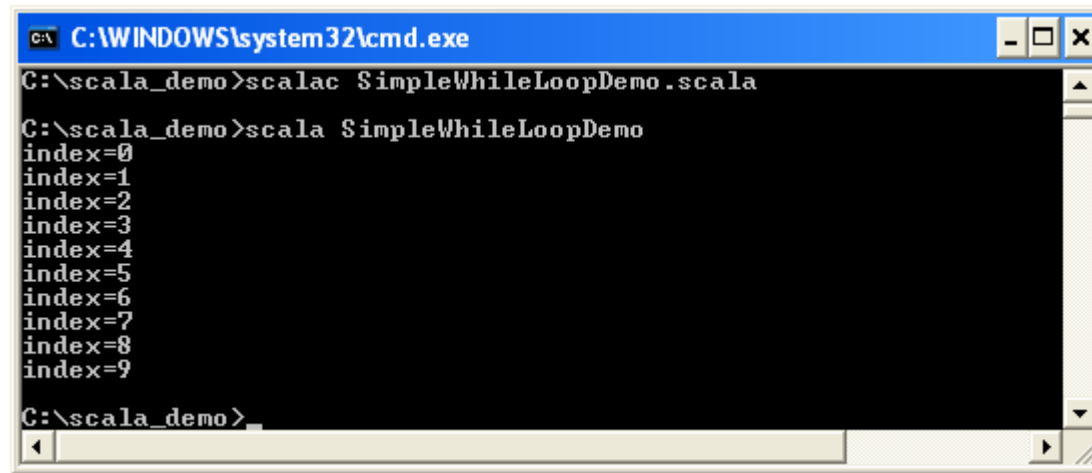
# The `while` Loop

- ❖ The `while` loop in Scala has the same known syntax used in Java.

A screenshot of a Notepad window titled "SimpleWhileLoopDemo.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The text area contains the following Scala code:

```
object SimpleWhileLoopDemo
{
    def main(args: Array[String])
    {
        var index: Int = 0
        while(index < 10)
        {
            println("index=" + index)
            index = index + 1
        }
    }
}
```

# The while Loop



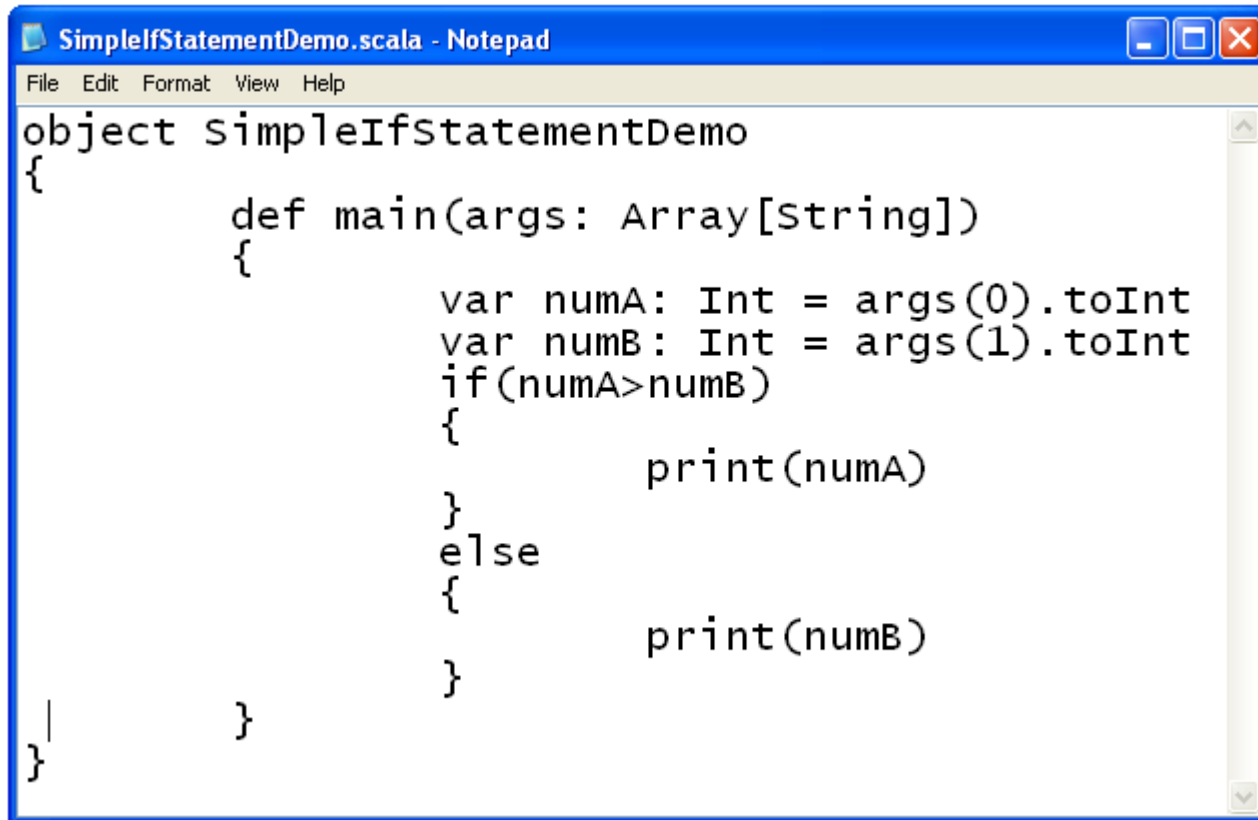
```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac SimpleWhileLoopDemo.scala
C:\scala_demo>scala SimpleWhileLoopDemo
index=0
index=1
index=2
index=3
index=4
index=5
index=6
index=7
index=8
index=9
C:\scala_demo>
```

The image shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "C:\scala\_demo>". The first command entered is "scalac SimpleWhileLoopDemo.scala", which compiles the Scala file. The second command is "scala SimpleWhileLoopDemo", which runs the program. The output of the program is a series of lines: "index=0", "index=1", "index=2", "index=3", "index=4", "index=5", "index=6", "index=7", "index=8", and "index=9". The prompt then returns to "C:\scala\_demo>".

# The `if` Statement

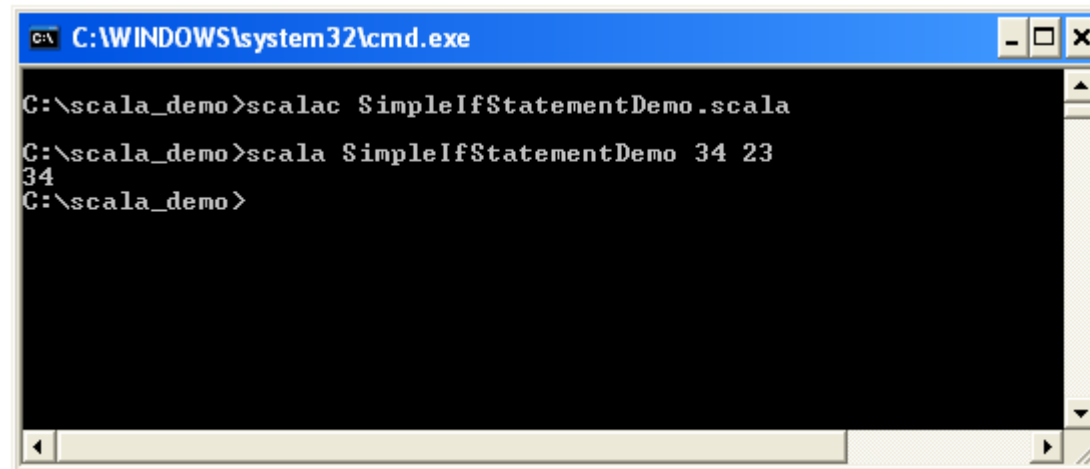
- ❖ The `if` statement in Scala uses the same known syntax from Java.

# The `if` Statement

A screenshot of a Notepad window titled "SimpleIfStatementDemo.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The code inside is written in Scala and demonstrates an if statement. It defines an object "SimpleIfStatementDemo" with a "main" method that takes an array of strings as arguments. Inside the "main" method, two integers, "numA" and "numB", are assigned from the first and second arguments. An if statement follows, checking if "numA" is greater than "numB". If true, it prints "numA"; otherwise, it prints "numB".

```
object SimpleIfStatementDemo
{
    def main(args: Array[String])
    {
        var numA: Int = args(0).toInt
        var numB: Int = args(1).toInt
        if(numA>numB)
        {
            print(numA)
        }
        else
        {
            print(numB)
        }
    }
}
```

# The `if` Statement



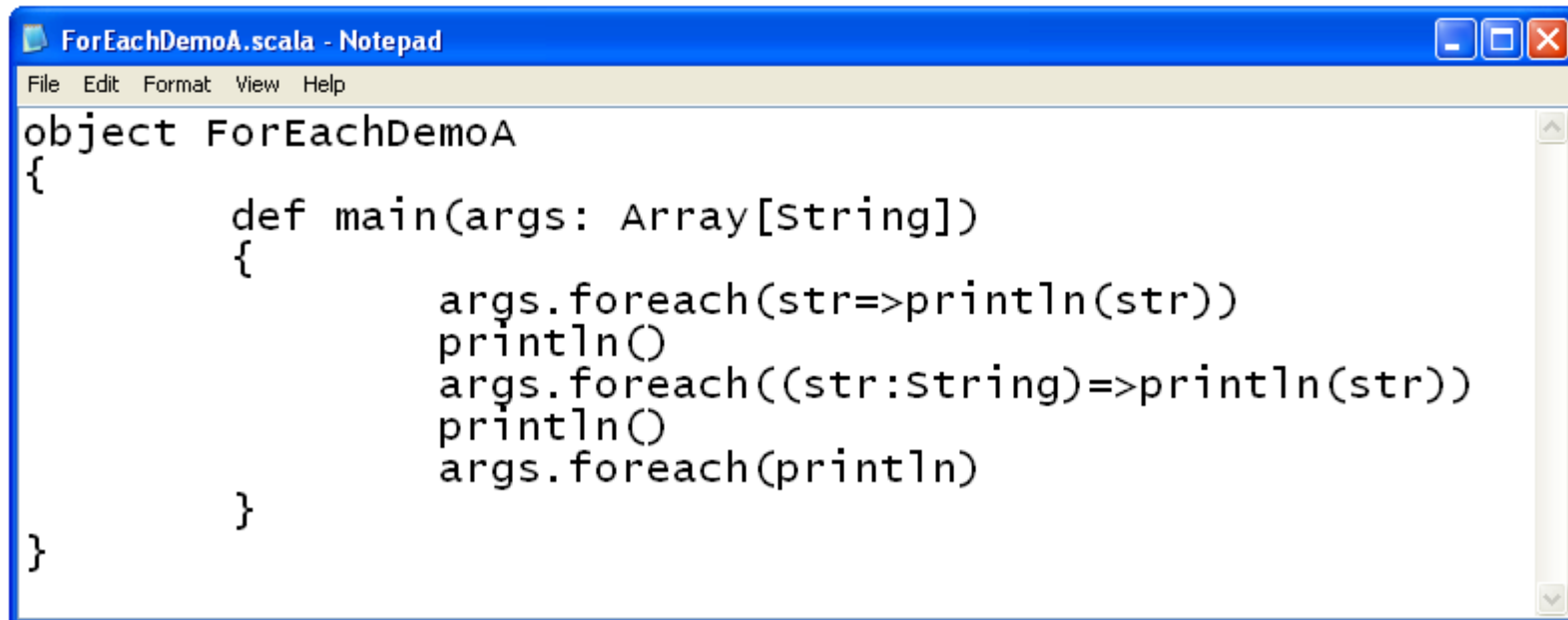
```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac SimpleIfStatementDemo.scala
C:\scala_demo>scala SimpleIfStatementDemo 34 23
34
C:\scala_demo>
```

A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\WINDOWS\system32\cmd.exe" along with standard window control buttons (minimize, maximize, close). The main area is black with white text. The text shows the following sequence of commands and output: a prompt "C:\scala\_demo>" followed by "scalac SimpleIfStatementDemo.scala", another prompt "C:\scala\_demo>" followed by "scala SimpleIfStatementDemo 34 23", and the output "34". The prompt "C:\scala\_demo>" appears again at the end.

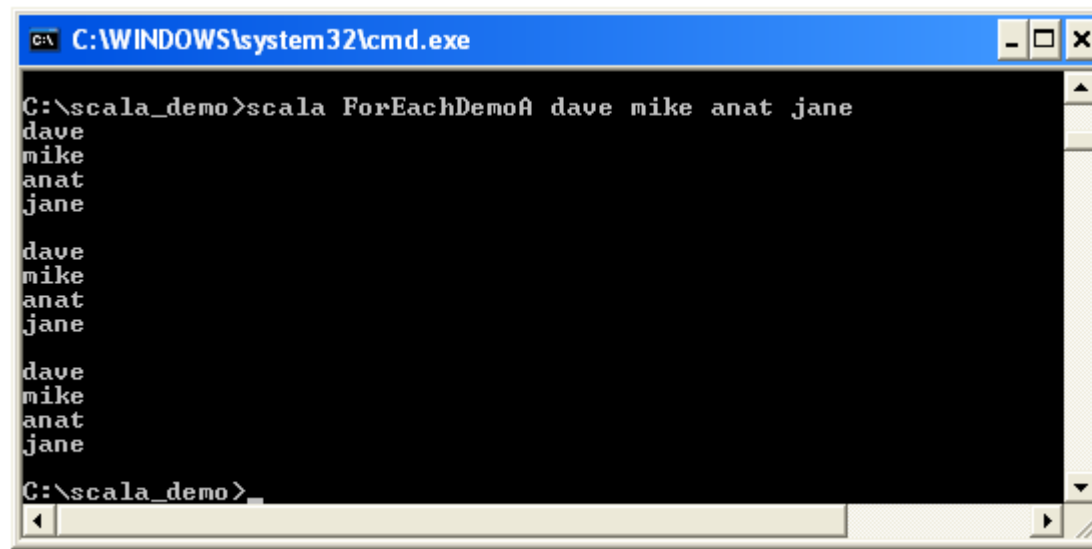
# The foreach Loop

- ❖ The `foreach` statement in Scala allows us iterating elements in array.



```
object ForEachDemoA
{
    def main(args: Array[String])
    {
        args.foreach(str=>println(str))
        println()
        args.foreach((str:String)=>println(str))
        println()
        args.foreach(println)
    }
}
```

# The foreach Loop



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scala ForEachDemoA dave mike anat jane
dave
mike
anat
jane

dave
mike
anat
jane

dave
mike
anat
jane

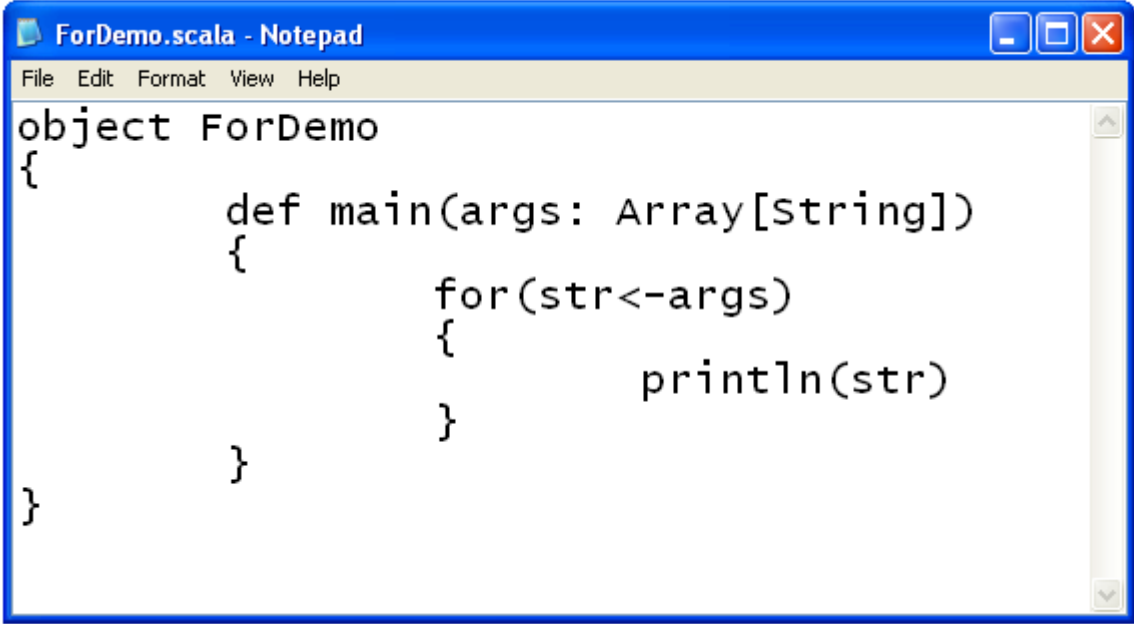
C:\scala_demo>
```

The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "C:\scala\_demo>". The user has entered the command "scala ForEachDemoA dave mike anat jane". The output of the program is displayed in three groups, each containing the names "dave", "mike", "anat", and "jane" on separate lines. The first group is followed by a blank line, and the second group is also followed by a blank line. The prompt "C:\scala\_demo>" is visible at the bottom of the window.



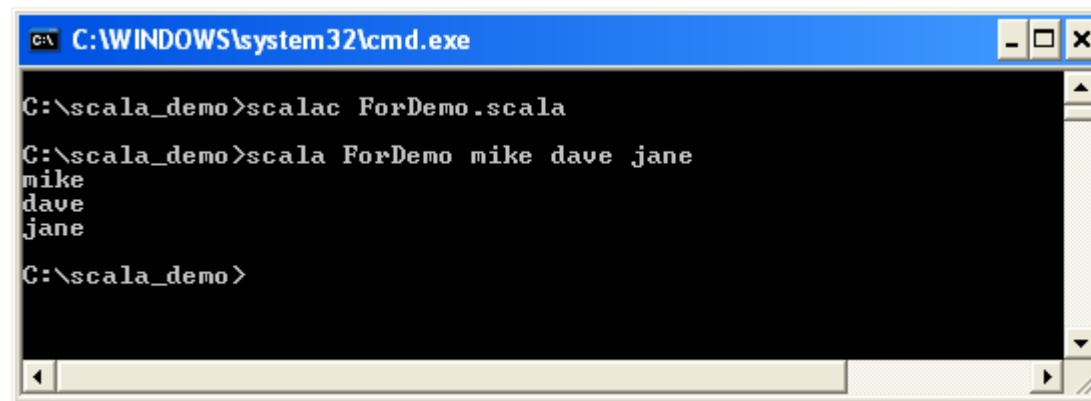
# The `for` Loop

- ❖ The `for` statement in Scala allows us iterating elements in array.



```
object ForDemo
{
    def main(args: Array[String])
    {
        for(str<-args)
        {
            println(str)
        }
    }
}
```

# The for Loop

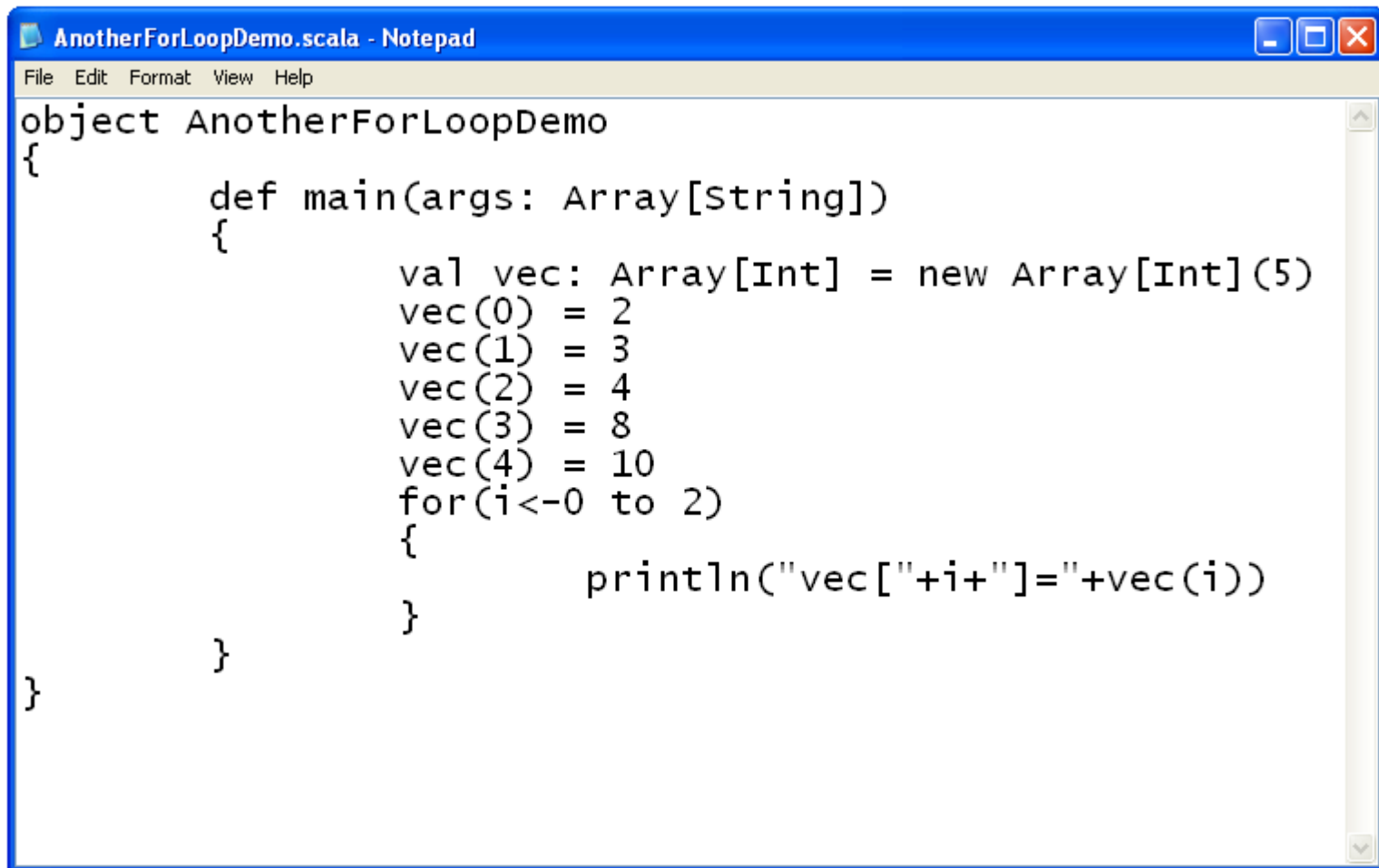


```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac ForDemo.scala
C:\scala_demo>scala ForDemo mike dave jane
mike
dave
jane
C:\scala_demo>
```

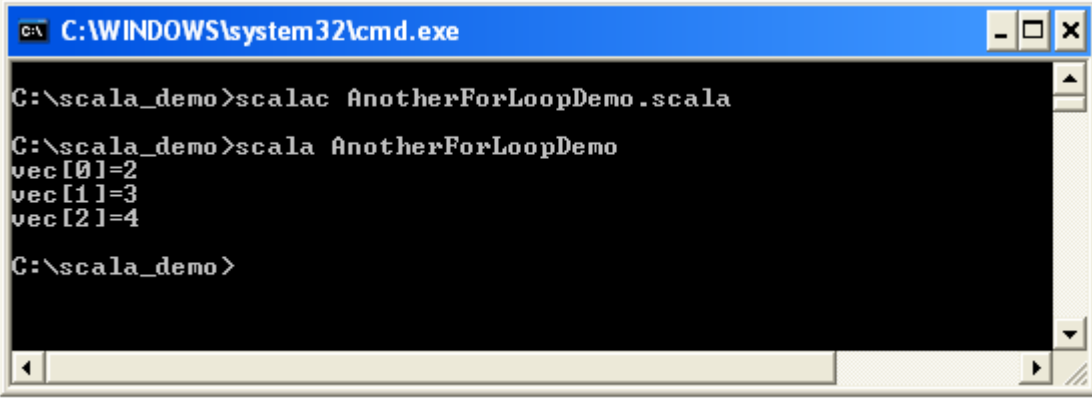
A screenshot of a Windows command prompt window. The title bar is blue and contains the text 'C:\WINDOWS\system32\cmd.exe'. The window has standard minimize, maximize, and close buttons. The main area is black with white text. The text shows the following sequence of commands and output: 'C:\scala\_demo>scalac ForDemo.scala', 'C:\scala\_demo>scala ForDemo mike dave jane', followed by three lines of output: 'mike', 'dave', and 'jane'. The prompt 'C:\scala\_demo>' appears again at the bottom.

# The for Loop

A screenshot of a Notepad window titled "AnotherForLoopDemo.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The code inside is as follows:

```
object AnotherForLoopDemo
{
    def main(args: Array[String])
    {
        val vec: Array[Int] = new Array[Int](5)
        vec(0) = 2
        vec(1) = 3
        vec(2) = 4
        vec(3) = 8
        vec(4) = 10
        for(i<-0 to 2)
        {
            println("vec["+i+"]="+vec(i))
        }
    }
}
```

# The for Loop



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac AnotherForLoopDemo.scala

C:\scala_demo>scala AnotherForLoopDemo
vec[0]=2
vec[1]=3
vec[2]=4

C:\scala_demo>
```

A screenshot of a Windows command prompt window. The title bar is blue and contains the text 'C:\WINDOWS\system32\cmd.exe'. The main area is black with white text. The text shows the following sequence of commands and output: 'C:\scala\_demo>scalac AnotherForLoopDemo.scala', 'C:\scala\_demo>scala AnotherForLoopDemo', 'vec[0]=2', 'vec[1]=3', 'vec[2]=4', and 'C:\scala\_demo>'. The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

# Scala Basic Types

- ❖ Scala supports the following basic types:

`Byte, Short, Int, Long, Char, String, Float, Double` and `Boolean`.

- ❖ Other than `String` that resides in `java.lang` package, all other types reside in the `scala` package (e.g. `scala.Int`).
- ❖ All members of the `scala` and the `java.lang` packages are automatically imported into every Scala source code.

# Scala Basic Types

- ❖ Scala basic types have the same range their equivalents in Java have.
- ❖ The syntax of the basic types literals is the same as in Java.

# Parametrize Arrays

- ❖ Scala allows us to instantiate classes using the `new` keyword.
- ❖ Creating a new object we can parametrize (configure) it with values and with types.

...

```
val vec = new Array[String](10)
```

...

```
val vec: Array[String] = new Array[String](10)
```

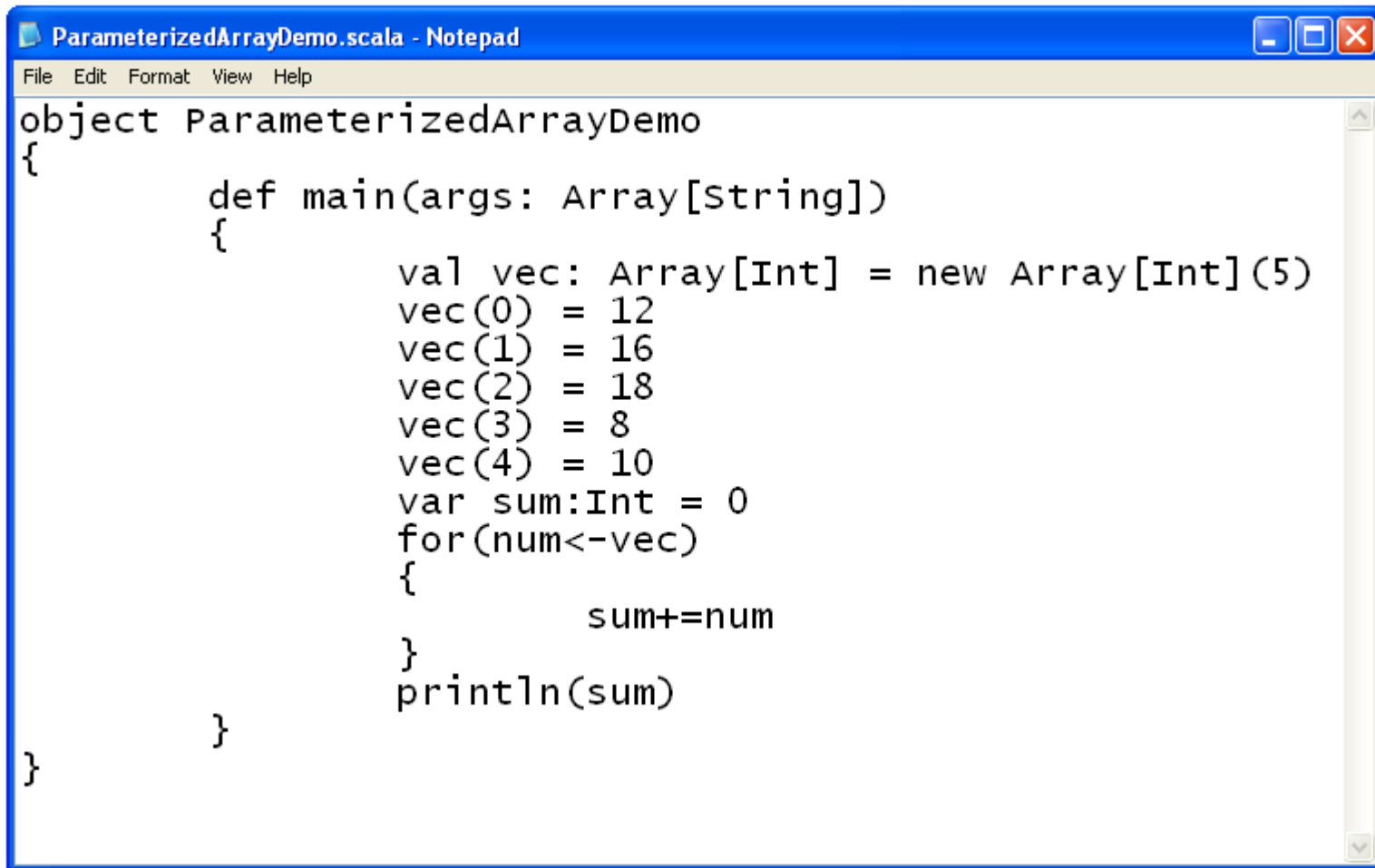
...

# Parametrize Arrays

- ❖ We parametrize with types by specifying them within square brackets. We parametrize with values by specifying them within parentheses.
- ❖ When we parametrize both with a value and with a type we specify the type first.

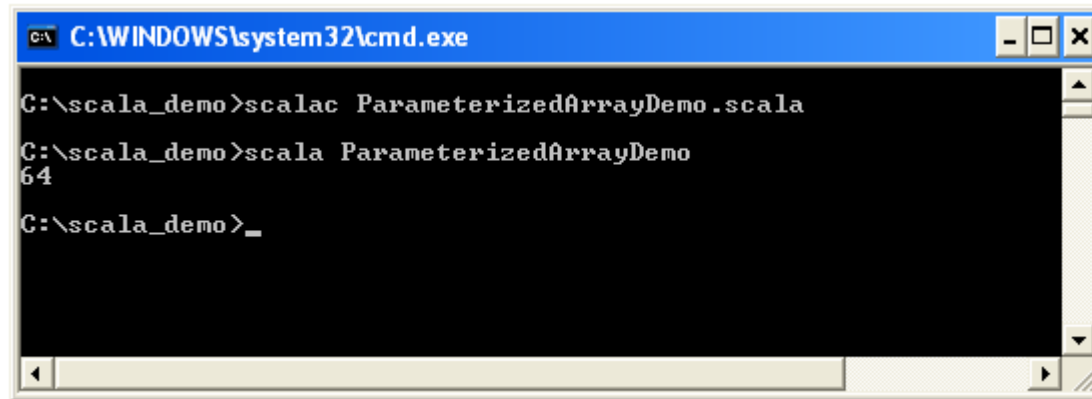


# Parametrize Arrays

A screenshot of a Notepad window titled "ParameterizedArrayDemo.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The code inside is a Scala object named "ParameterizedArrayDemo" with a "main" method. The "main" method takes an "Array[String]" as an argument and creates an "Array[Int]" named "vec" with 5 elements. It then iterates over the elements of "vec" and calculates their sum, which is printed at the end.

```
object ParameterizedArrayDemo
{
    def main(args: Array[String])
    {
        val vec: Array[Int] = new Array[Int](5)
        vec(0) = 12
        vec(1) = 16
        vec(2) = 18
        vec(3) = 8
        vec(4) = 10
        var sum:Int = 0
        for(num<-vec)
        {
            sum+=num
        }
        println(sum)
    }
}
```

# Parametrize Arrays



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac ParameterizedArrayDemo.scala
C:\scala_demo>scala ParameterizedArrayDemo
64
C:\scala_demo>_
```

The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The window contains the following text: "C:\scala\_demo>scalac ParameterizedArrayDemo.scala", "C:\scala\_demo>scala ParameterizedArrayDemo", "64", and "C:\scala\_demo>\_". The window has a standard Windows interface with a blue title bar, minimize, maximize, and close buttons, and a scroll bar on the right side.

# Arrays Implementation

- ❖ When referring an array element, the Scala compiler transform our code `vec(i)` into `vec.apply(i)`.
- ❖ Accessing an array element is in fact calling a method.  
When accessing an array element in order to change it, our call will be transformed into a call to the `vec.update(i, val)` method.

# Arrays Short Syntax

- ❖ We can create new arrays using the following abbreviated syntax.

...

```
val vec = Array("jane", "mike", "dave")
```

...

# Lists

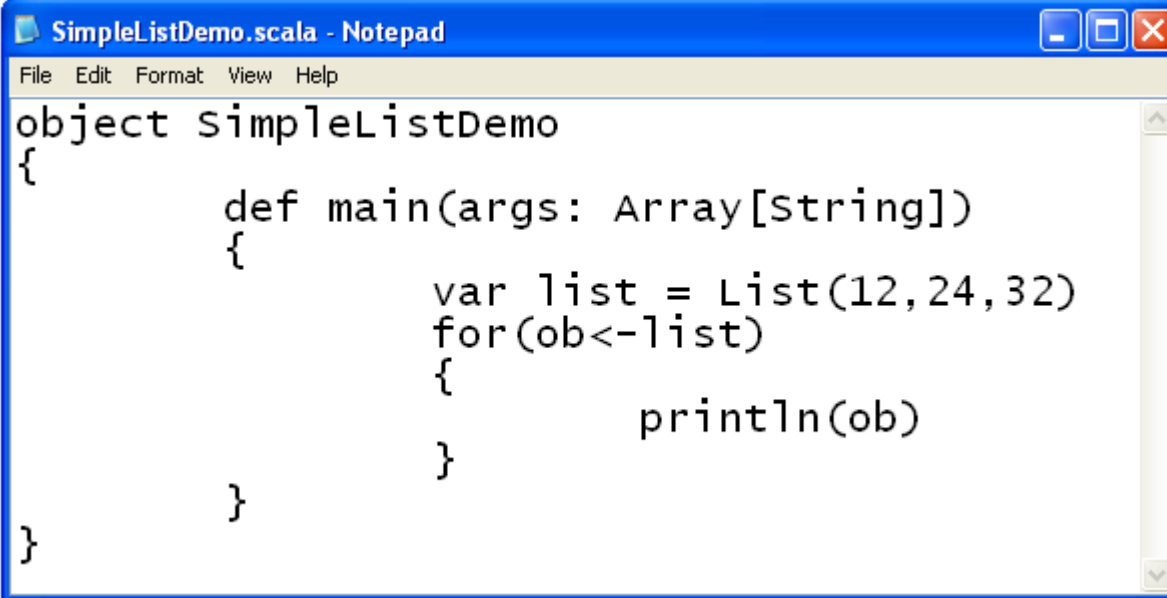
- ❖ The Scala `List` class represents an immutable sequence of objects that share the same type.

...

```
var list = List(23,12,534)
```

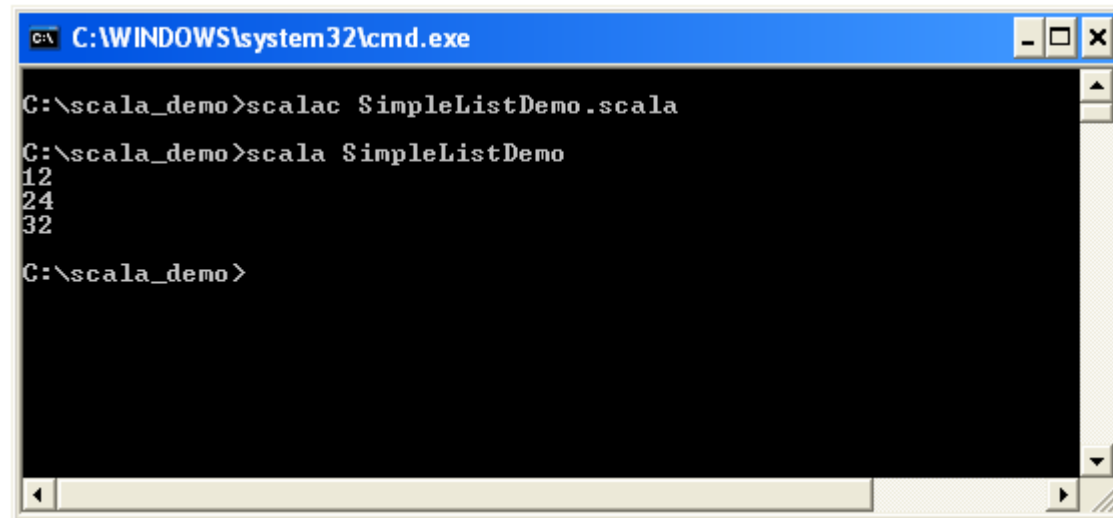
...

# Lists



```
object SimpleListDemo
{
    def main(args: Array[String])
    {
        var list = List(12, 24, 32)
        for(ob<-list)
        {
            println(ob)
        }
    }
}
```

# Lists

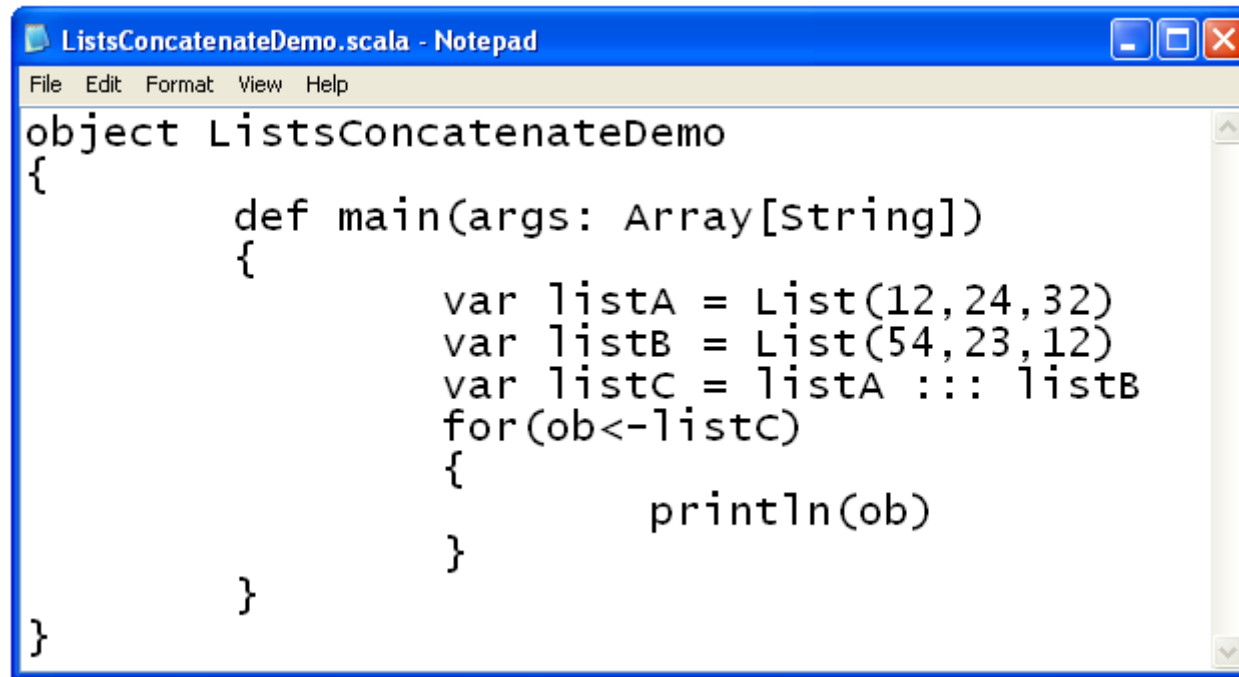


```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac SimpleListDemo.scala
C:\scala_demo>scala SimpleListDemo
12
24
32
C:\scala_demo>
```

# The :: Method

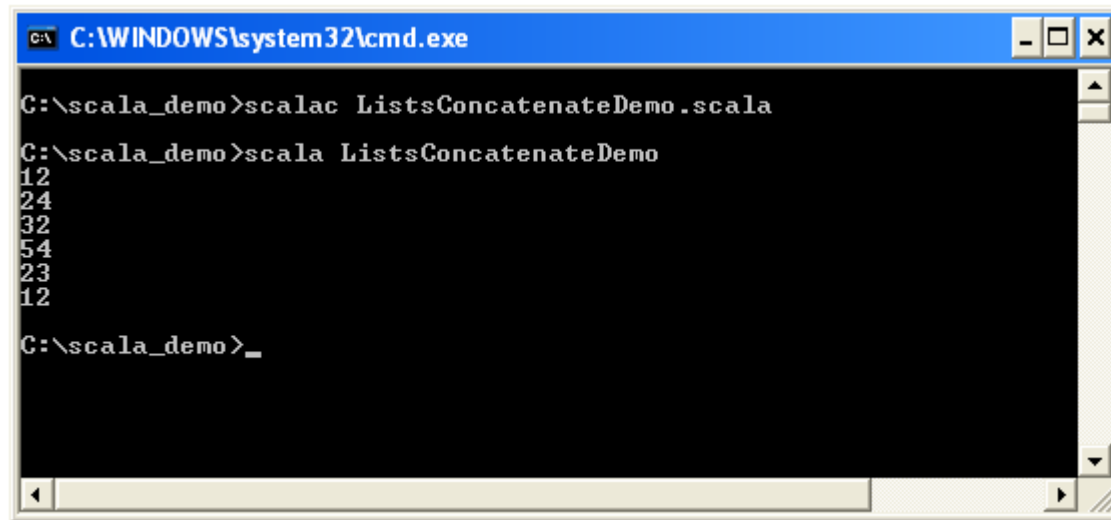
- ❖ The :: method concatenates two separated lists into a new one.

A screenshot of a Notepad window titled "ListsConcatenateDemo.scala - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The code inside is as follows:

```
object ListsConcatenateDemo
{
    def main(args: Array[String])
    {
        var listA = List(12,24,32)
        var listB = List(54,23,12)
        var listC = listA :: listB
        for(ob<-listC)
        {
            println(ob)
        }
    }
}
```



# The :: Method



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac ListsConcatenateDemo.scala

C:\scala_demo>scala ListsConcatenateDemo
12
24
32
54
23
12

C:\scala_demo>_
```

The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The window contains the following text:

```
C:\scala_demo>scalac ListsConcatenateDemo.scala

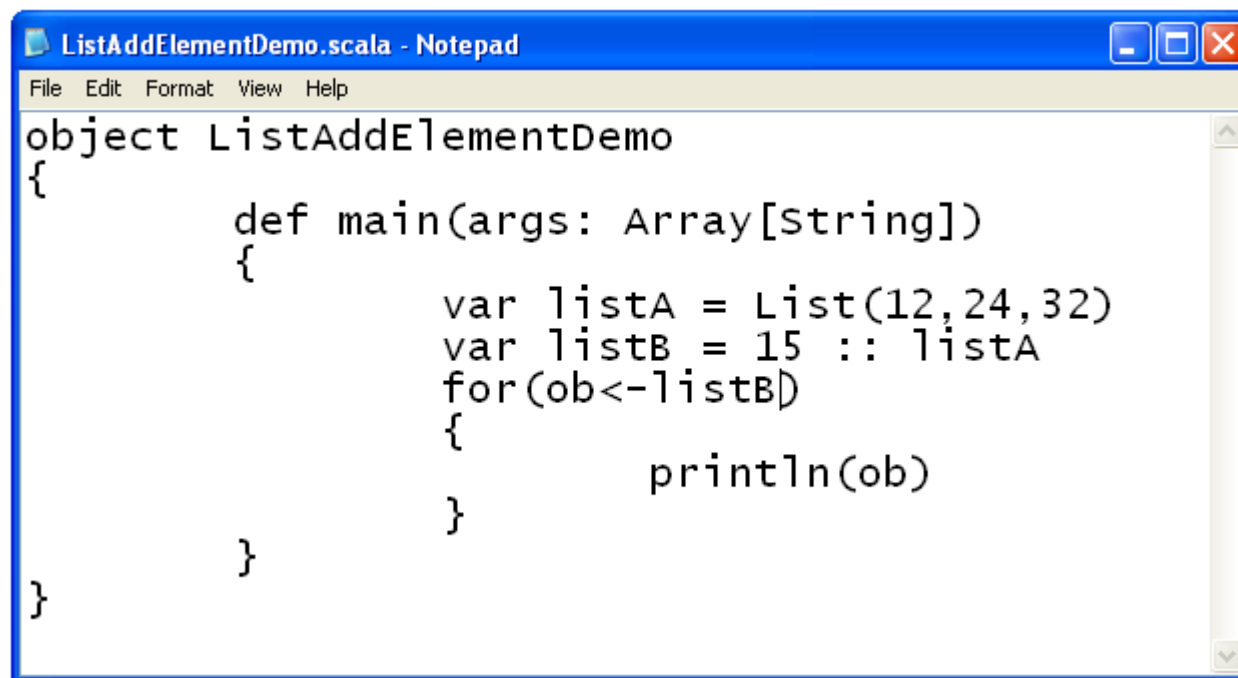
C:\scala_demo>scala ListsConcatenateDemo
12
24
32
54
23
12

C:\scala_demo>_
```

The output of the Scala program is a list of numbers: 12, 24, 32, 54, 23, and 12, each on a new line.

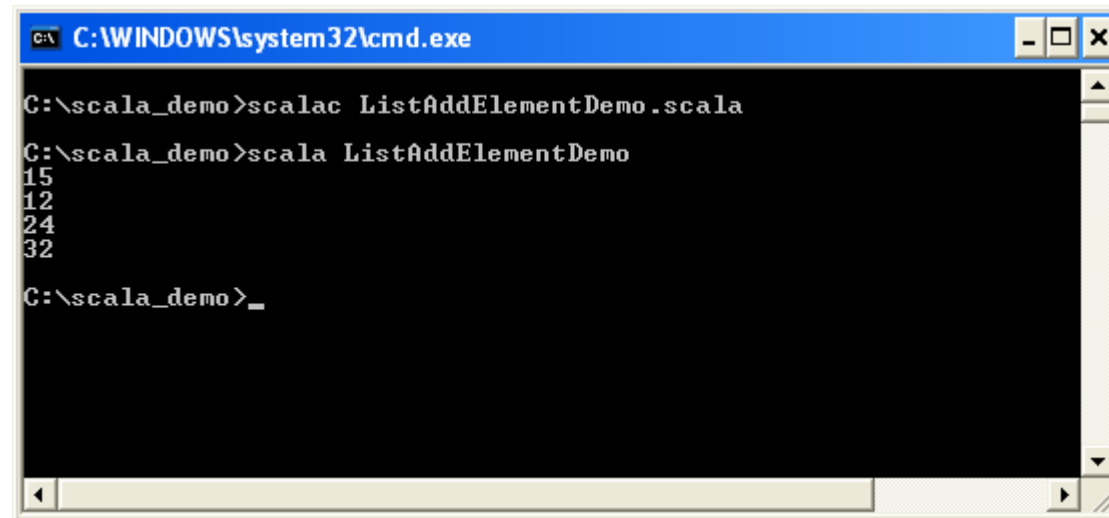
# The :: Method

- ❖ The :: method adds a new element to our list.



```
object ListAddElementDemo
{
    def main(args: Array[String])
    {
        var listA = List(12,24,32)
        var listB = 15 :: listA
        for(ob<-listB)
        {
            println(ob)
        }
    }
}
```

# The :: Method



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac ListAddElementDemo.scala

C:\scala_demo>scala ListAddElementDemo
15
12
24
32

C:\scala_demo>_
```

The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The window contains the following text:   
C:\scala\_demo>scalac ListAddElementDemo.scala   
C:\scala\_demo>scala ListAddElementDemo   
15   
12   
24   
32   
C:\scala\_demo>\_

# The Nil Value

- ❖ The `Nil` special value is Scala equivalent to an empty list.

# Tuples

- ❖ Tuples are immutable collections. Unlike List, a Tuple can contain elements of different types.
- ❖ We can access a Tuple's elements using the '.' operator together with the '\_' underscore and the element index number.

...

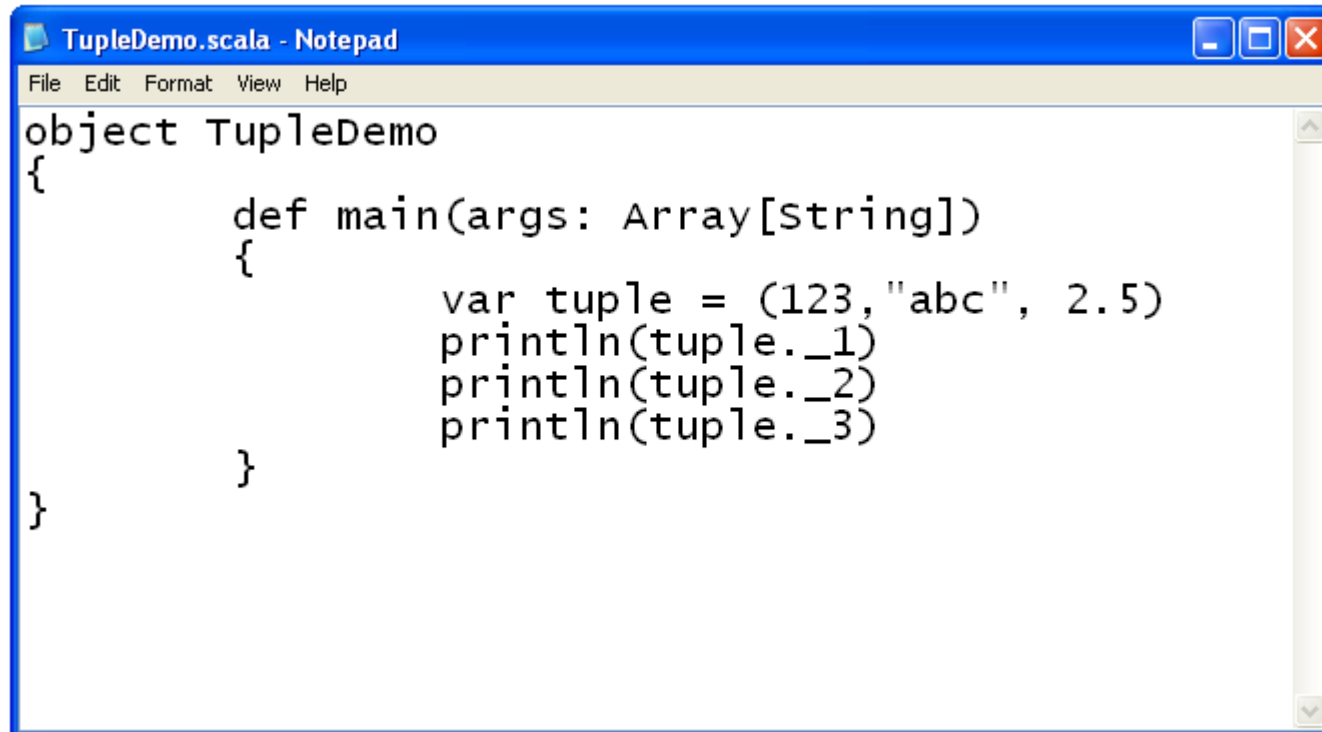
```
var ob = (123, "abc")
```

```
println(ob._1)
```

```
println(ob._2)
```

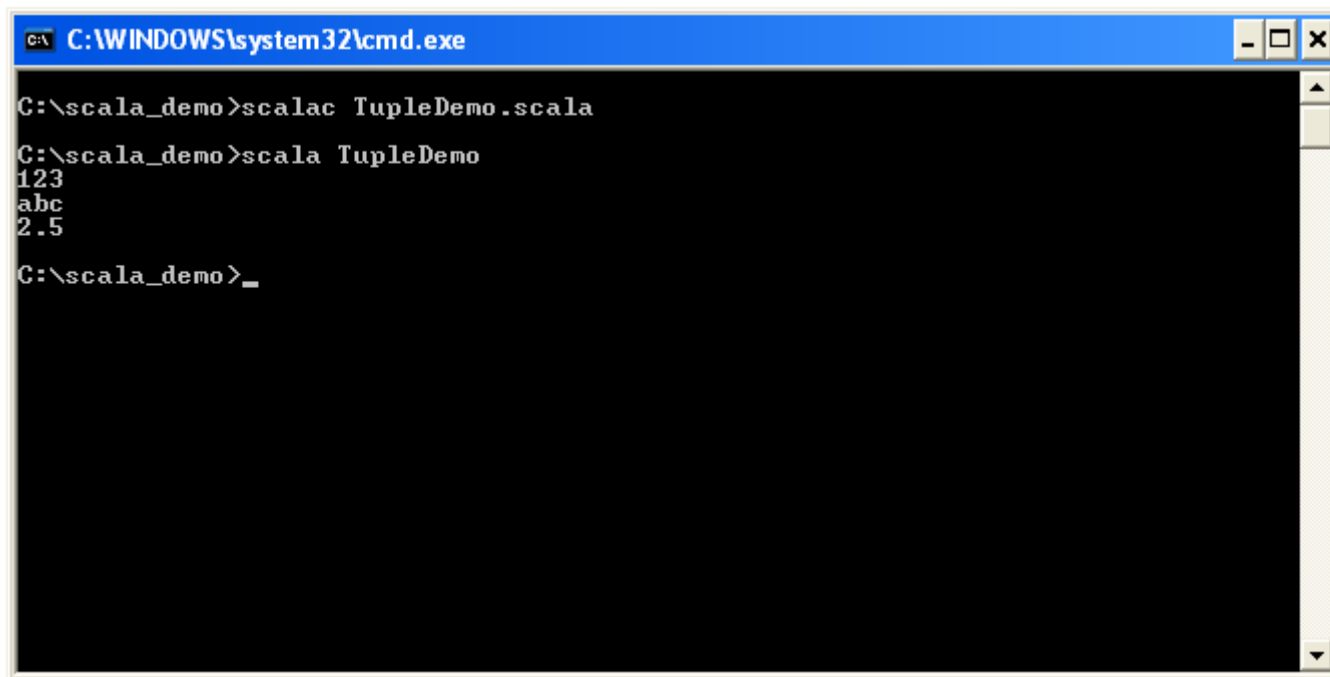
...

# Tuples



```
object TupleDemo
{
    def main(args: Array[String])
    {
        var tuple = (123, "abc", 2.5)
        println(tuple._1)
        println(tuple._2)
        println(tuple._3)
    }
}
```

# Tuples



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac TupleDemo.scala
C:\scala_demo>scala TupleDemo
123
abc
2.5
C:\scala_demo>_
```

# Tuples

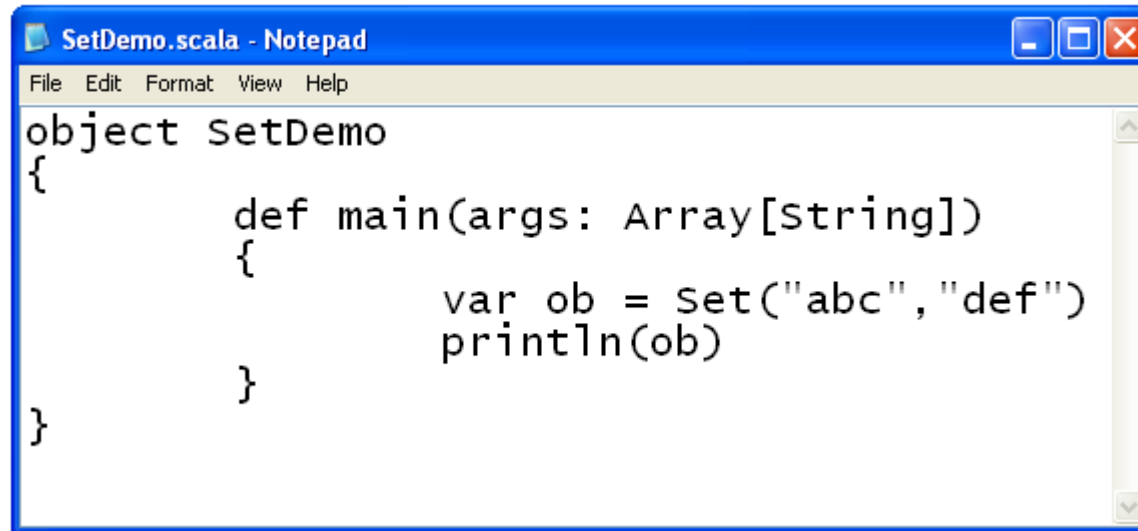
- ❖ The actual type of a tuple depends on the number of elements it contains and the type of each one of them.



# Sets

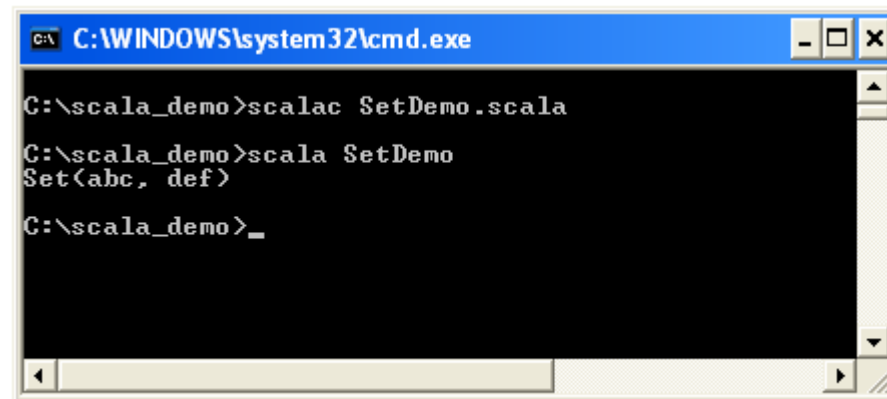
- ❖ Set is a collection of elements. Elements cannot repeat themselves.

# Sets



```
object SetDemo
{
    def main(args: Array[String])
    {
        var ob = Set("abc", "def")
        println(ob)
    }
}
```

# Sets



```
C:\> C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac SetDemo.scala

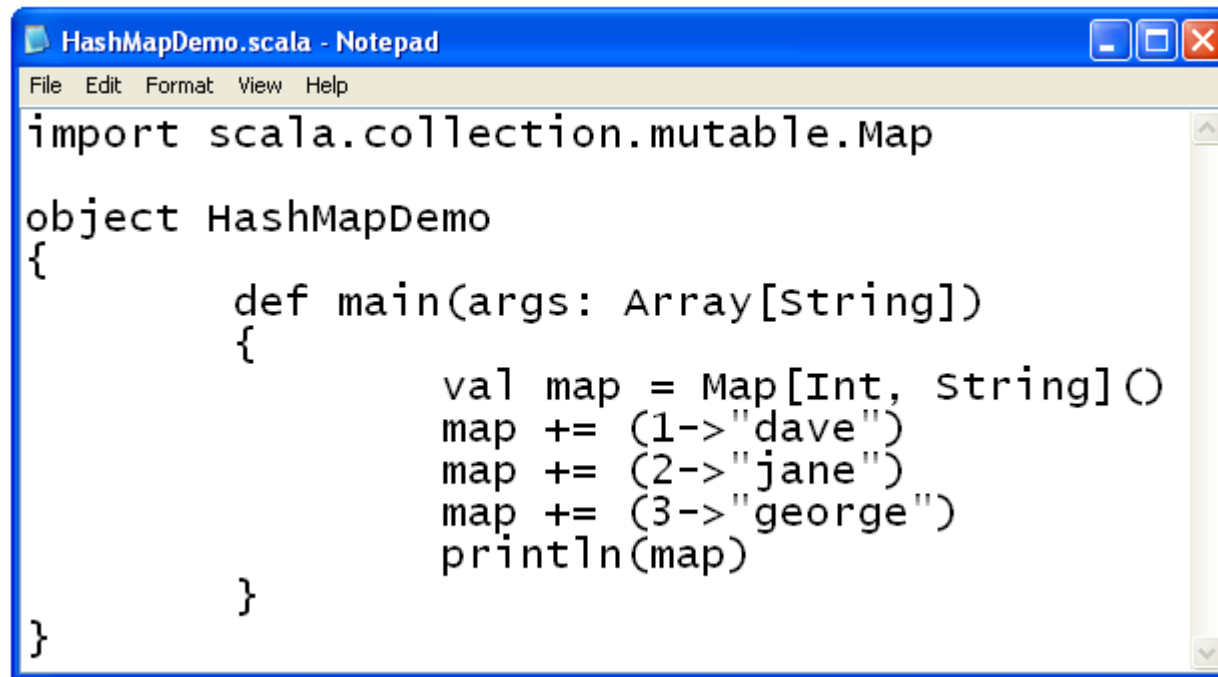
C:\scala_demo>scala SetDemo
Set(abc, def)

C:\scala_demo>_
```

The image shows a screenshot of a Windows command prompt window. The title bar at the top reads "C:\> C:\WINDOWS\system32\cmd.exe". The window contains the following text: "C:\scala\_demo>scalac SetDemo.scala", "C:\scala\_demo>scala SetDemo", "Set(abc, def)", and "C:\scala\_demo>\_". The window has a standard Windows interface with a blue title bar, a black command area, and a scroll bar on the right.

# Maps

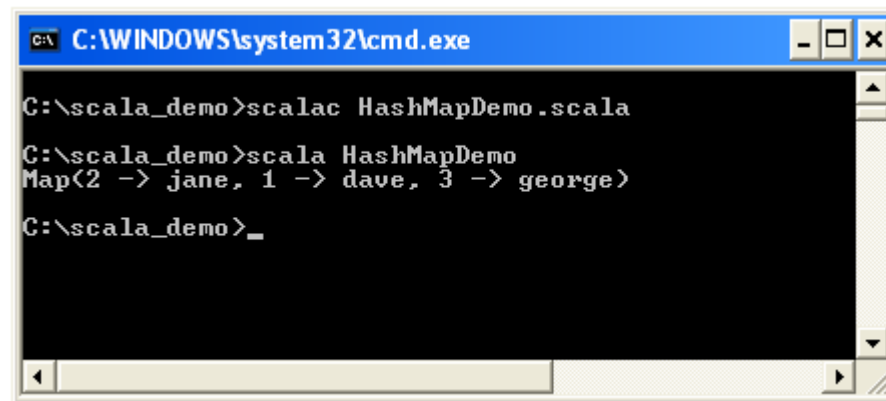
- ❖ Map holds key value pairs. Each key is unique. We cannot have the same key twice.



```
HashMapDemo.scala - Notepad
File Edit Format View Help
import scala.collection.mutable.Map

object HashMapDemo
{
    def main(args: Array[String])
    {
        val map = Map[Int, String]()
        map += (1->"dave")
        map += (2->"jane")
        map += (3->"george")
        println(map)
    }
}
```

# Maps



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scalac HashMapDemo.scala

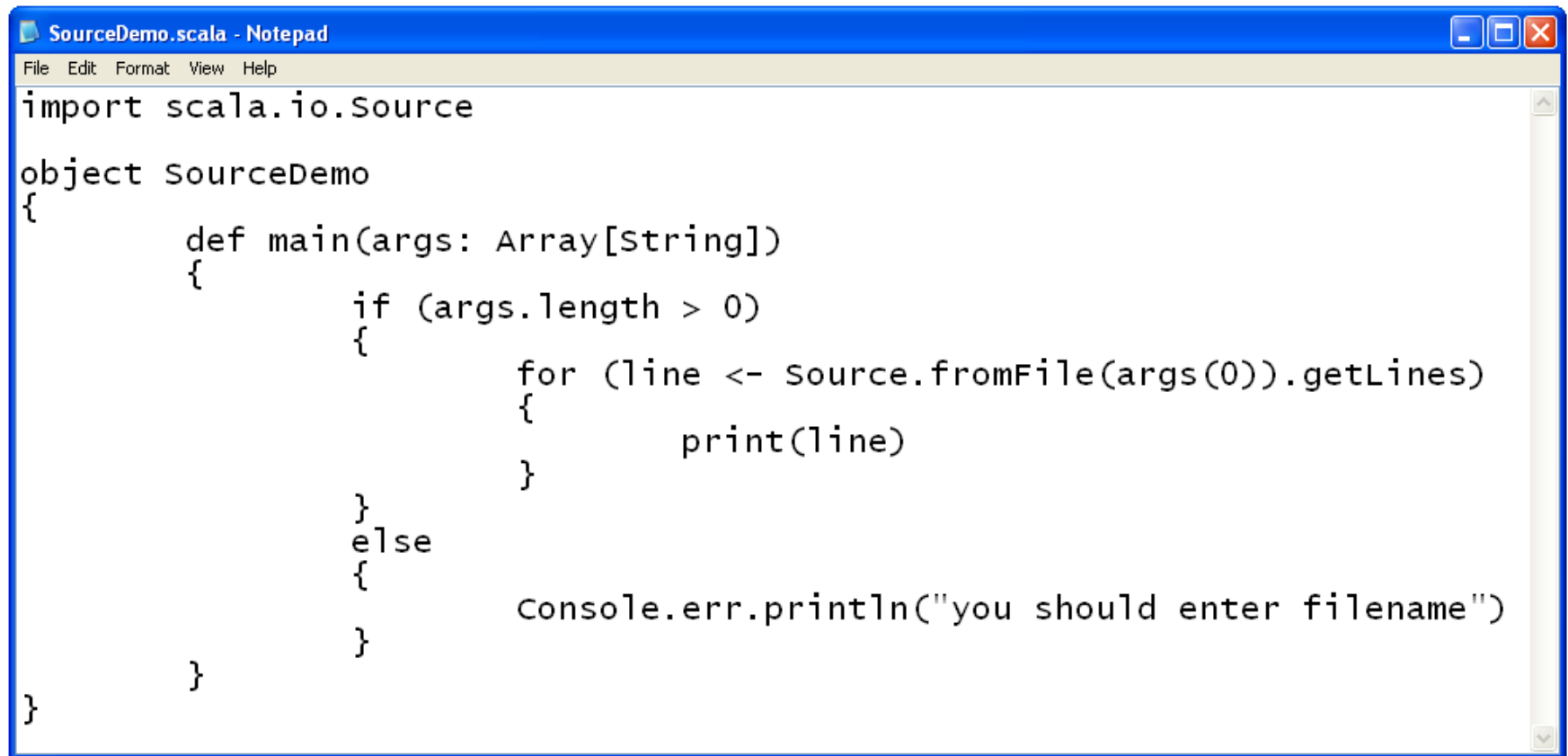
C:\scala_demo>scala HashMapDemo
Map(2 -> jane, 1 -> dave, 3 -> george)

C:\scala_demo>_
```

The image shows a Windows command prompt window with a blue title bar that reads "C:\WINDOWS\system32\cmd.exe". The window contains the following text: "C:\scala\_demo>scalac HashMapDemo.scala", "C:\scala\_demo>scala HashMapDemo", "Map(2 -> jane, 1 -> dave, 3 -> george)", and "C:\scala\_demo>\_". The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

# File Read

- ❖ Using the `Source` object we can easily read from a file.

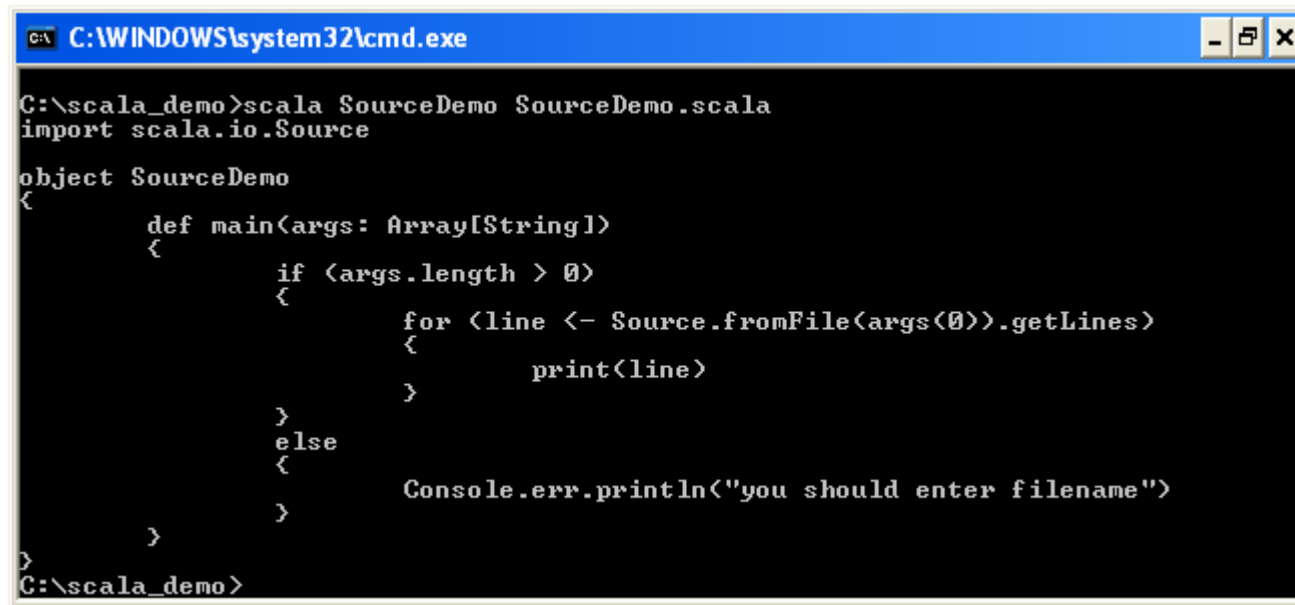


```
SourceDemo.scala - Notepad
File Edit Format View Help

import scala.io.Source

object SourceDemo
{
    def main(args: Array[String])
    {
        if (args.length > 0)
        {
            for (line <- Source.fromFile(args(0)).getLines)
            {
                print(line)
            }
        }
        else
        {
            Console.err.println("you should enter filename")
        }
    }
}
```

# File Read



```
C:\WINDOWS\system32\cmd.exe

C:\scala_demo>scala SourceDemo SourceDemo.scala
import scala.io.Source

object SourceDemo
{
    def main(args: Array[String])
    {
        if (args.length > 0)
        {
            for (line <- Source.fromFile(args(0)).getLines)
            {
                print(line)
            }
        }
        else
        {
            Console.err.println("you should enter filename")
        }
    }
}

C:\scala_demo>
```

# Two Dimension Arrays

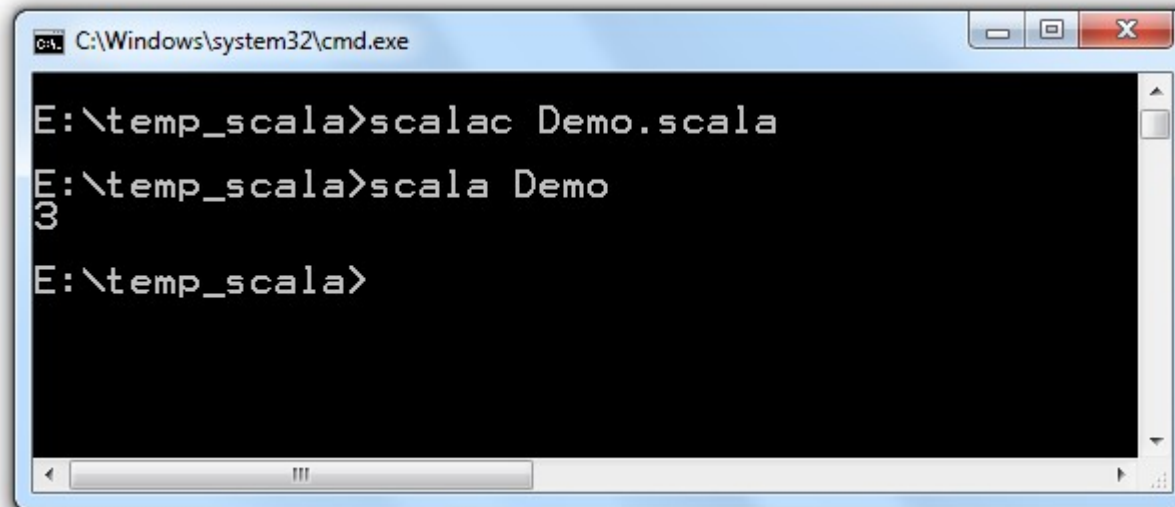
- ❖ We can easily create two dimension arrays by referring each one of the first array cells as one that holds another array.



# Two Dimension Arrays

```
object Demo extends Application
{
    val matrix = Array(Array(4,3,2),Array(5,6,2))
    println(matrix(0)(1))
}
```

# Two Dimension Arrays



```
C:\Windows\system32\cmd.exe

E:\temp_scala>scalac Demo.scala
E:\temp_scala>scala Demo
3
E:\temp_scala>
```

The image shows a Windows command prompt window with a blue title bar. The title bar text is "C:\Windows\system32\cmd.exe". The window contains three lines of text: "E:\temp\_scala>scalac Demo.scala", "E:\temp\_scala>scala Demo", and "E:\temp\_scala>". The output of the second command is the number "3". The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

# Blocks in Scala

- ❖ Blocks in Scala are simple braces that surround code.

```
{  
  ...  
}
```

- ❖ The definitions within a block are visible from within the block only.
- ❖ The definitions within a block shadow definitions with the same names that belong to the block's outer scope.

# Blocks in Scala

- ❖ The block has a value just as any other expression. Its value is the value of the last expression it includes.

```
object Program
{
  def main(args: Array[String]):Unit =
  {
    val a = 4
    def func(num:Int) = num*num
    val number =
    {
      val b = func(4)
      val a = 6
      a+b
    } + 2
    println(number)
  }
}
```



# Multiple Lines Expressions

- ❖ When the expression spans over multiple lines its evaluation might be different from what we would have meant. The evaluation might exclude parts of the expression.

```
object Program
{
  def main(args: Array[String]):Unit =
  {
    var numberA:Int = 14
    var numberB:Int = 8
    var numberC:Int = numberA
    + numberB
    println(numberC)
  }
}
```



# Lazy Evaluation

- ❖ Using the `lazy` keyword we can specify that we want to have a lazy evaluation for the expression we assign a `val` variable with.

# Lazy Evaluation

```
object LazyDemo {  
  def main(args:Array[String]):Unit = {  
    var a = 3  
    var b = 4  
    lazy val c = a+b  
    a = 10  
    b = 40  
    println(c)  
  }  
}
```

