

Control Statements

Introduction

- ❖ Scala supports the following built in control structures: `if`, `while`, `for`, `try` **and** `match`.

The Return Value

- ❖ Most of Scala's supported control statements return a value.
- ❖ This behavior is common to functional programming languages. This behavior assists us shorten the code.

The `if` Expression

- ❖ Just as in many other programming languages, if the boolean expression is true then code branch is executed.

```
if (condition)
{
    ...
}
```

The `if` Expression

- ❖ The `if` expression returns a value which is either the first expression or the second. We can use that for assigning an `if else` expression into a variable.

...

```
var name = if(num>0) "canada" else "israel"
```

...

The `while` Loop

- ❖ The `while` loop works the same as in other software programming languages.

...

```
while(condition)
```

```
{
```

```
    //do something
```

```
}
```

...

The do..while Loop

- ❖ The do while loop works the same as in other software programming languages.

```
...
```

```
do
```

```
{
```

```
    //do something
```

```
}
```

```
while(condition)
```

```
...
```

The `for` Expression

- ❖ The `for` expression allows us to use it in several ways.
- ❖ The simplest way is iterating through all elements of a given collection.

...

```
for(str <- vec)
{
    println(str)
}
```

...

The `for` Expression

- ❖ The `for` expression can also work on range of values.

...

```
for(i<- 1 to 8)
```

```
{
```

```
  println("i="+i)
```

```
}
```

...

The `for` Expression

- ❖ We can iterate till the upper bound (included) using the `to` keyword.

...

```
for(i<- 1 to 8)
```

```
{
```

```
  println("i="+i)
```

```
}
```

...

The `for` Expression

- ❖ We can iterate till the upper bound (excluded) using the

`until` keyword.

...

```
for(i<- 1 until 8)
```

```
{
```

```
  println("i="+i)
```

```
}
```

...

The `for` Expression

- ❖ Adding the filter expression we can iterate all values excluding those that don't meet the condition the filter sets.

...

```
val vec = Array(1,2,3,4,5,6,7,8,9,10)
```

```
for(i<- vec if(i%2==0))
```

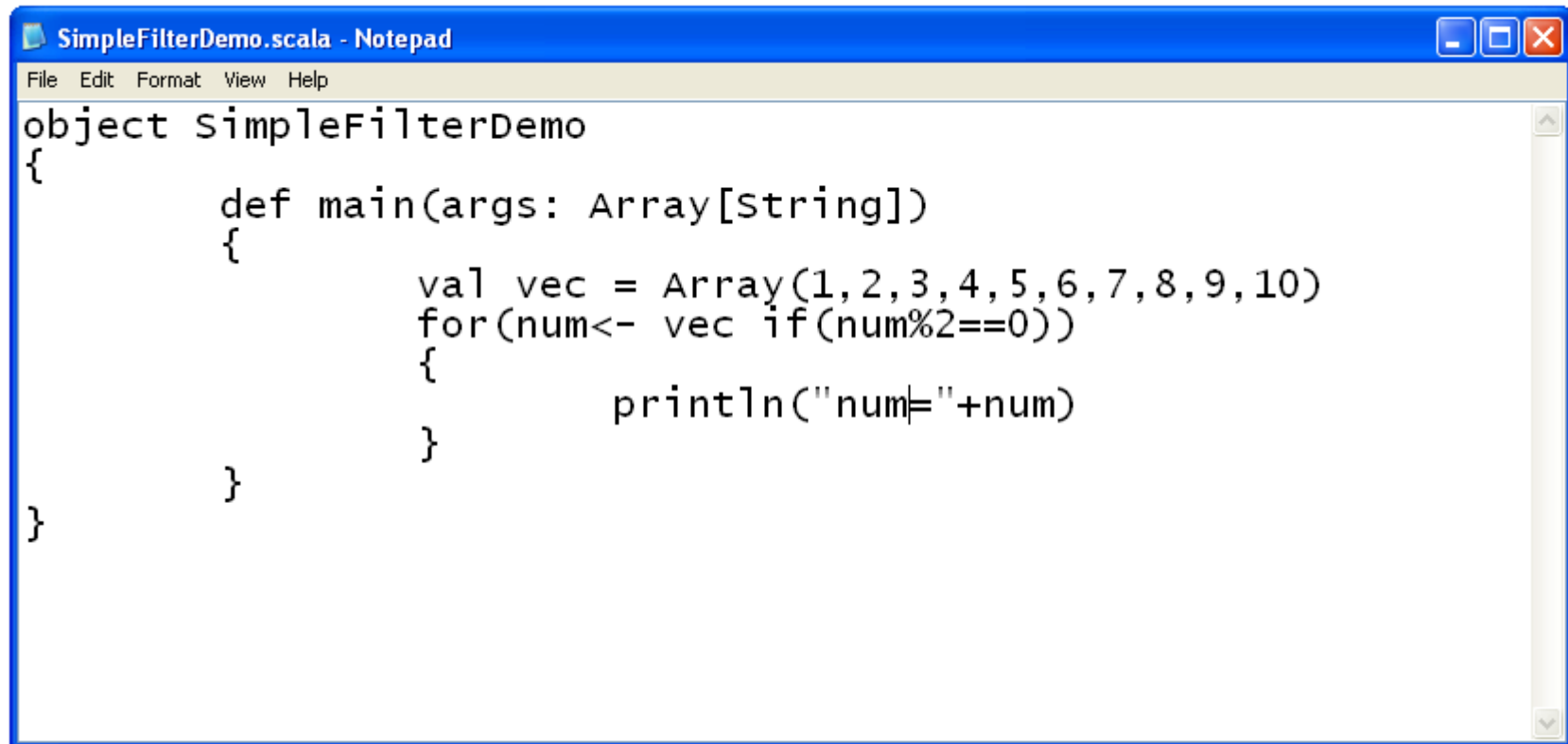
```
{
```

```
    println("i="+i)
```

```
}
```

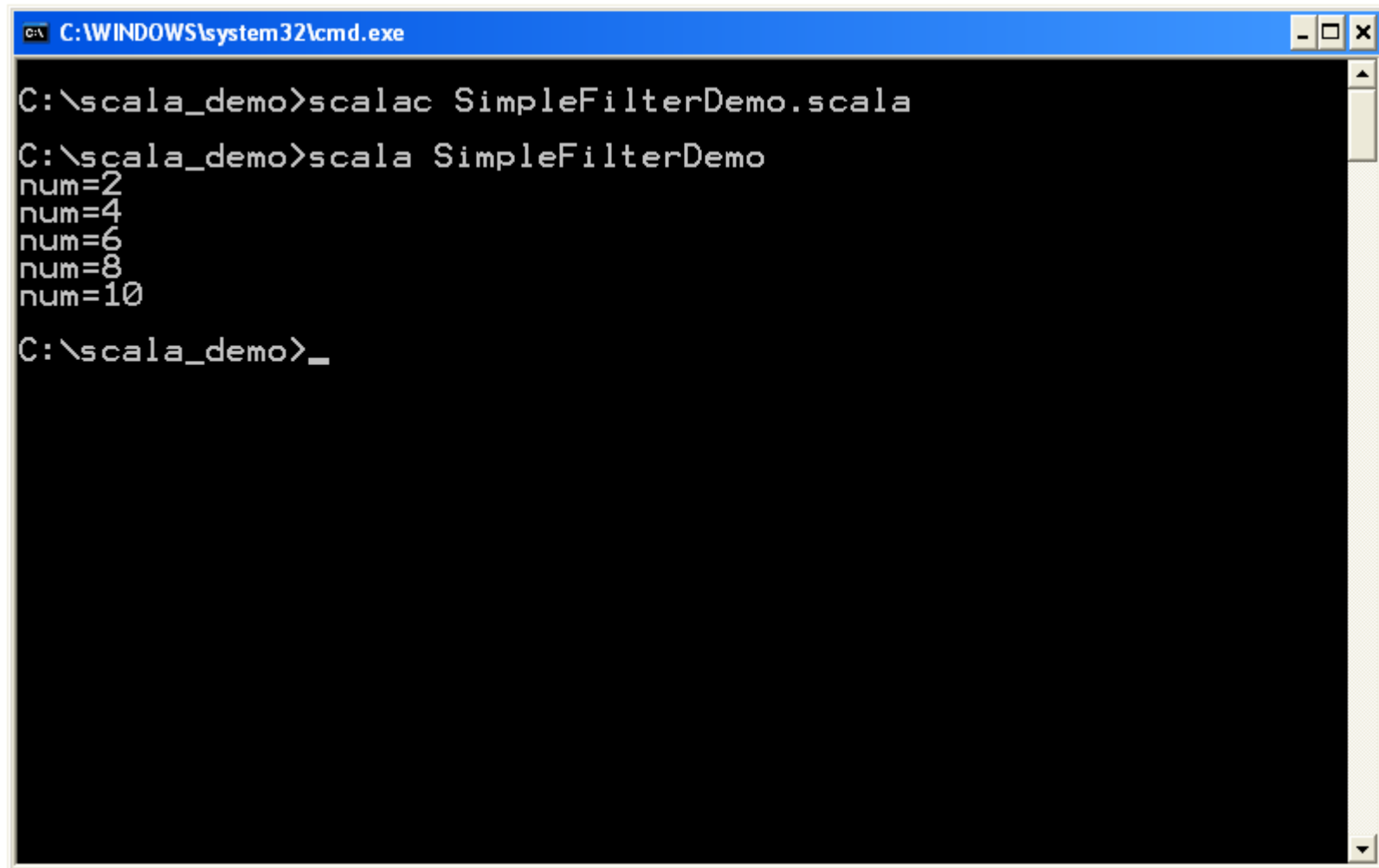
...

The `for` Expression



```
SimpleFilterDemo.scala - Notepad
File Edit Format View Help
object SimpleFilterDemo
{
    def main(args: Array[String])
    {
        val vec = Array(1,2,3,4,5,6,7,8,9,10)
        for(num<- vec if(num%2==0))
        {
            println("num="+num)
        }
    }
}
```

The `for` Expression



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac SimpleFilterDemo.scala
C:\scala_demo>scala SimpleFilterDemo
num=2
num=4
num=6
num=8
num=10
C:\scala_demo>_
```

The `for` Expression

- ❖ Complex filters conditions composed of several separated conditions are feasible.

The `for` Expression

- ❖ We can add multiple `<-` clauses in order to get nested loops.

...

```
var rows = Array(1,2,3,4,5,6,7,8,9,10)
```

```
var cols = Array(1,2,3,4,5,6,7,8,9,10)
```

```
for(row <- rows)
```

```
{
```

```
    for(col <- cols)
```

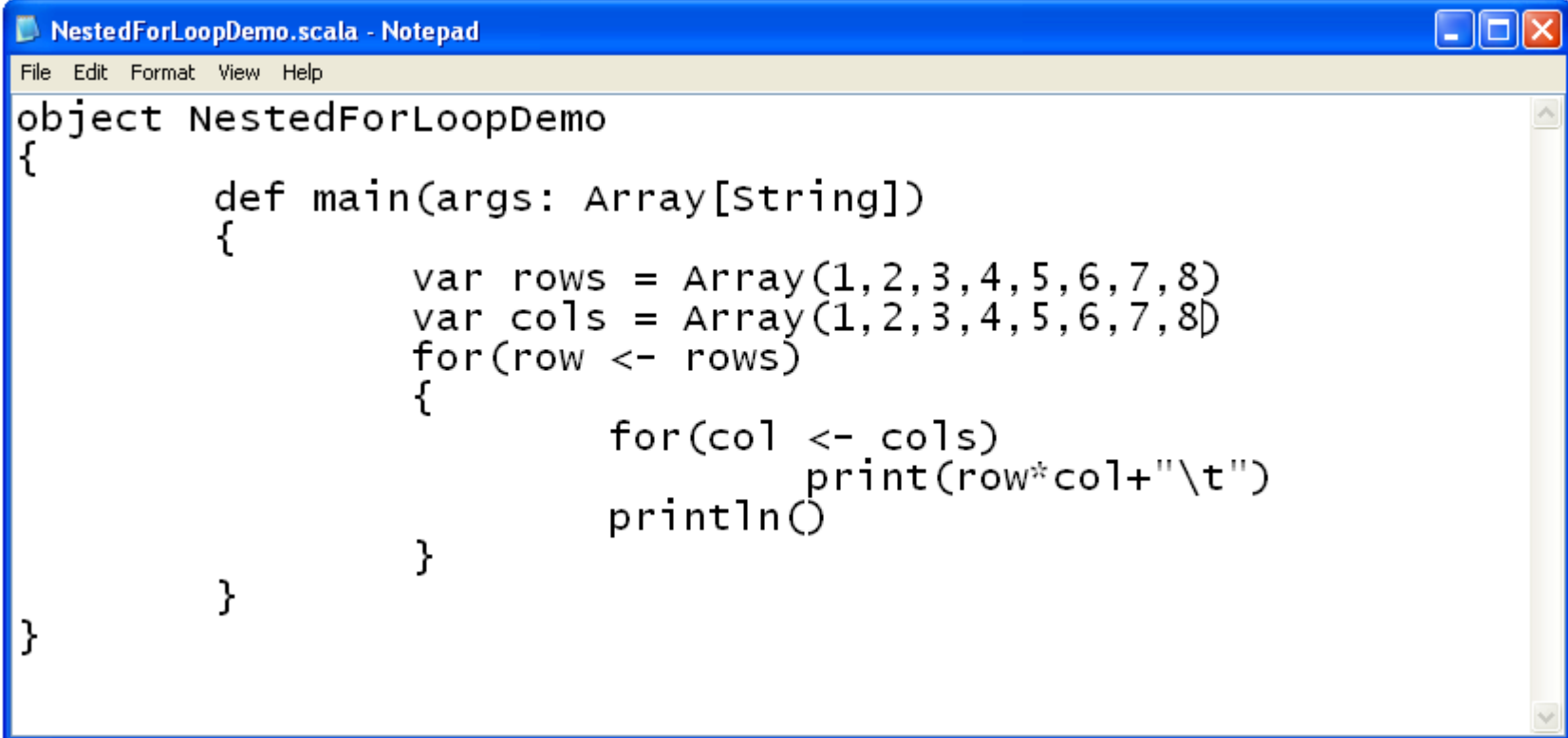
```
        print(row*col+"\t")
```

```
    println()
```

```
}
```

...

The `for` Expression



```
NestedForLoopDemo.scala - Notepad
File Edit Format View Help
object NestedForLoopDemo
{
  def main(args: Array[String])
  {
    var rows = Array(1,2,3,4,5,6,7,8)
    var cols = Array(1,2,3,4,5,6,7,8)
    for(row <- rows)
    {
      for(col <- cols)
      print(row*col+"\t")
      println()
    }
  }
}
```

The `for` Expression

```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac NestedForLoopDemo.scala
C:\scala_demo>scala NestedForLoopDemo
1      2      3      4      5      6      7      8
2      4      6      8      10     12     14     16
3      6      9      12     15     18     21     24
4      8      12     16     20     24     28     32
5      10     15     20     25     30     35     40
6      12     18     24     30     36     42     48
7      14     21     28     35     42     49     56
8      16     24     32     40     48     56     64
C:\scala_demo>
```

The `for` Expression

- ❖ We can alternatively add multiple `<-` clauses within the same brackets.

...

```
var rows = Array(1,2,3,4,5,6,7,8)
```

```
var cols = Array(1,2,3,4,5,6,7,8)
```

```
var sum: Int = 0
```

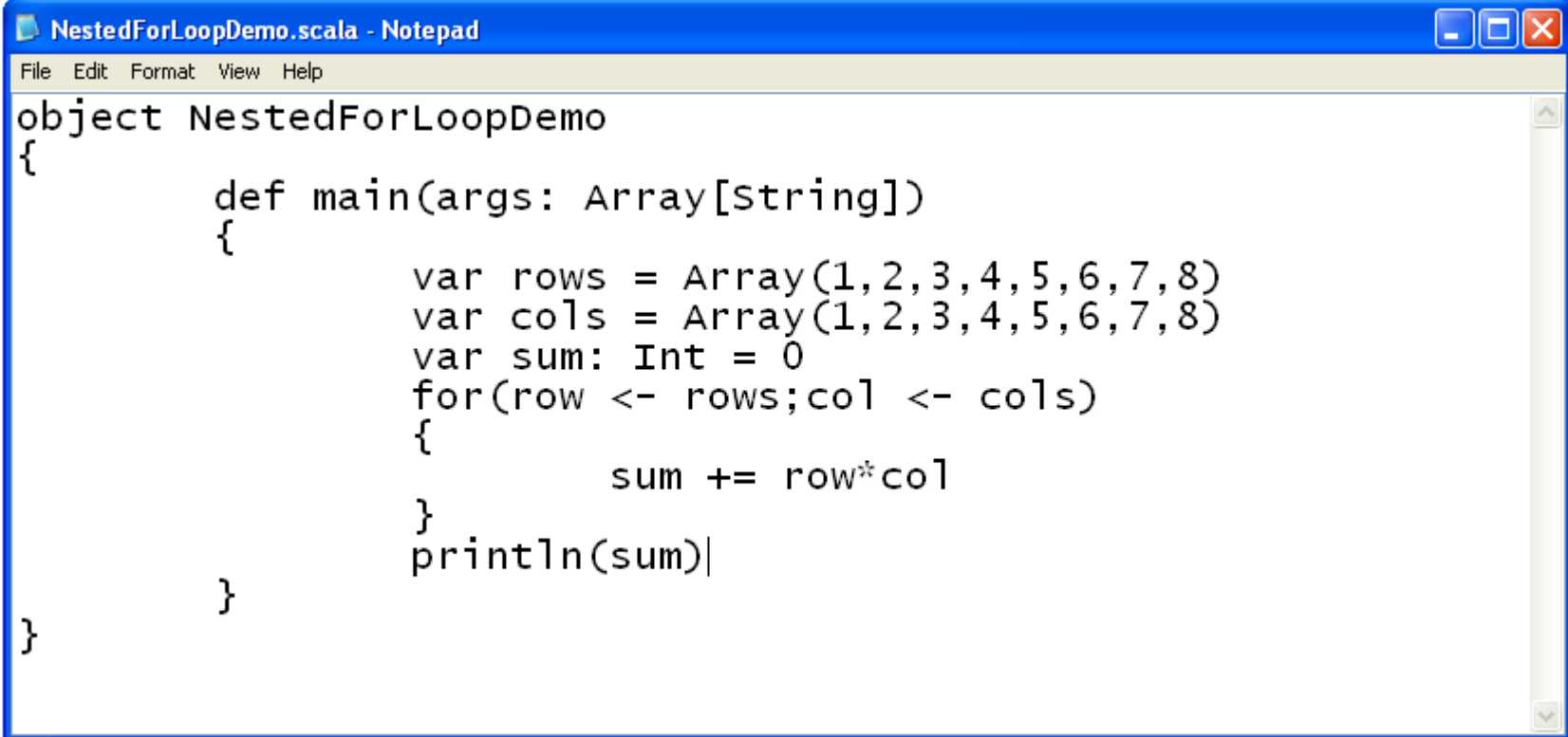
```
for(row <- rows; col <- cols)
```

```
    sum += row*col
```

```
println(sum)
```

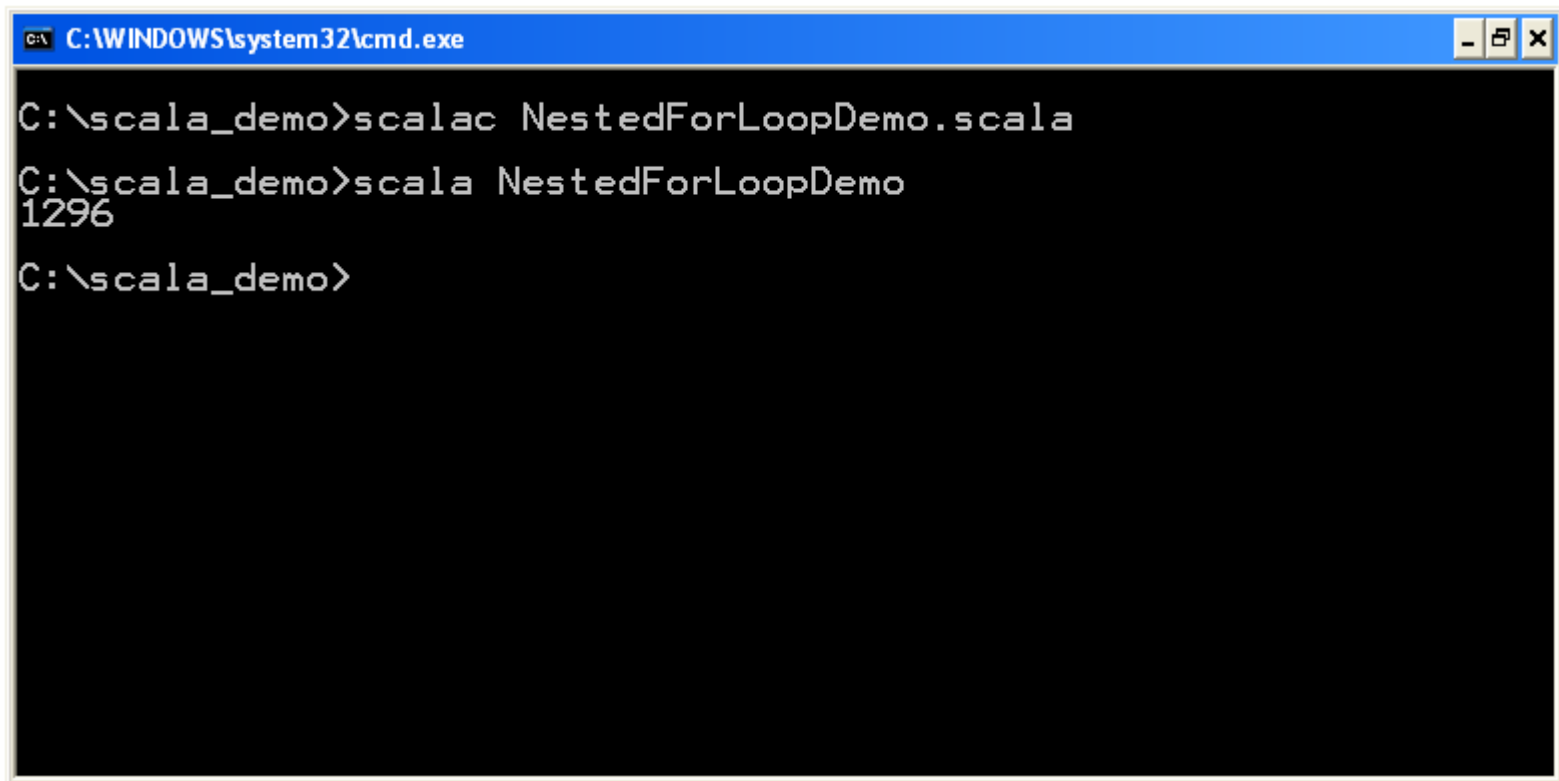
...

The `for` Expression

A screenshot of a Notepad window titled "NestedForLoopDemo.scala - Notepad". The window contains Scala code that demonstrates a nested for loop. The code defines an object named "NestedForLoopDemo" with a "main" method. Inside the "main" method, two arrays of integers from 1 to 8 are created: "rows" and "cols". A variable "sum" of type "Int" is initialized to 0. A nested for loop iterates over each element in "rows" and "cols", calculating the product of the current row and column values and adding it to "sum". Finally, the value of "sum" is printed to the console.

```
object NestedForLoopDemo
{
    def main(args: Array[String])
    {
        var rows = Array(1,2,3,4,5,6,7,8)
        var cols = Array(1,2,3,4,5,6,7,8)
        var sum: Int = 0
        for(row <- rows; col <- cols)
        {
            sum += row*col
        }
        println(sum)
    }
}
```

The `for` Expression



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac NestedForLoopDemo.scala
C:\scala_demo>scala NestedForLoopDemo
1296
C:\scala_demo>
```

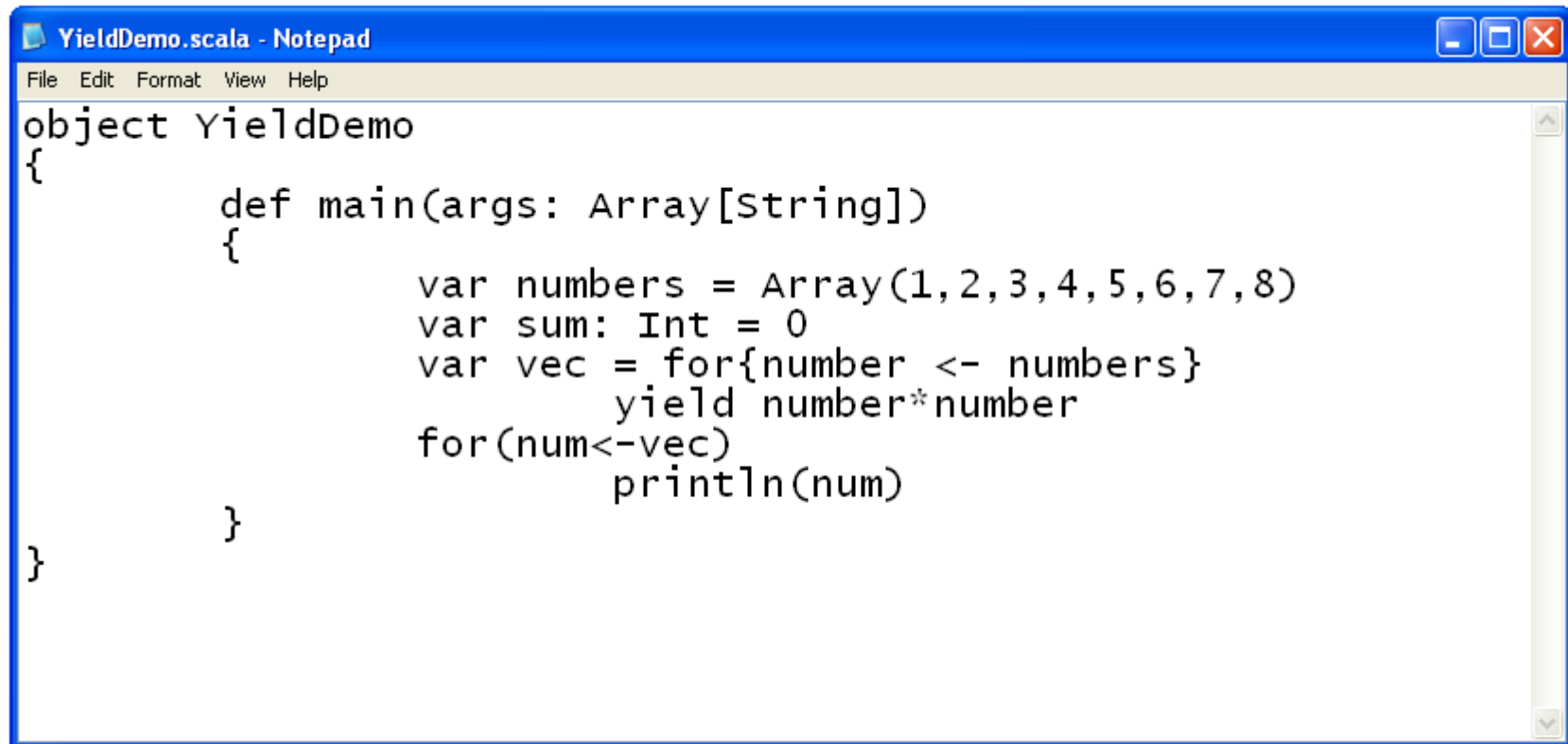
The `for-yield` Expression

- ❖ We can iterate a given collection and generate a new one based on the elements we iterate.
- ❖ The syntax of the `for-yield` expression is

```
for{clauses} yield body
```

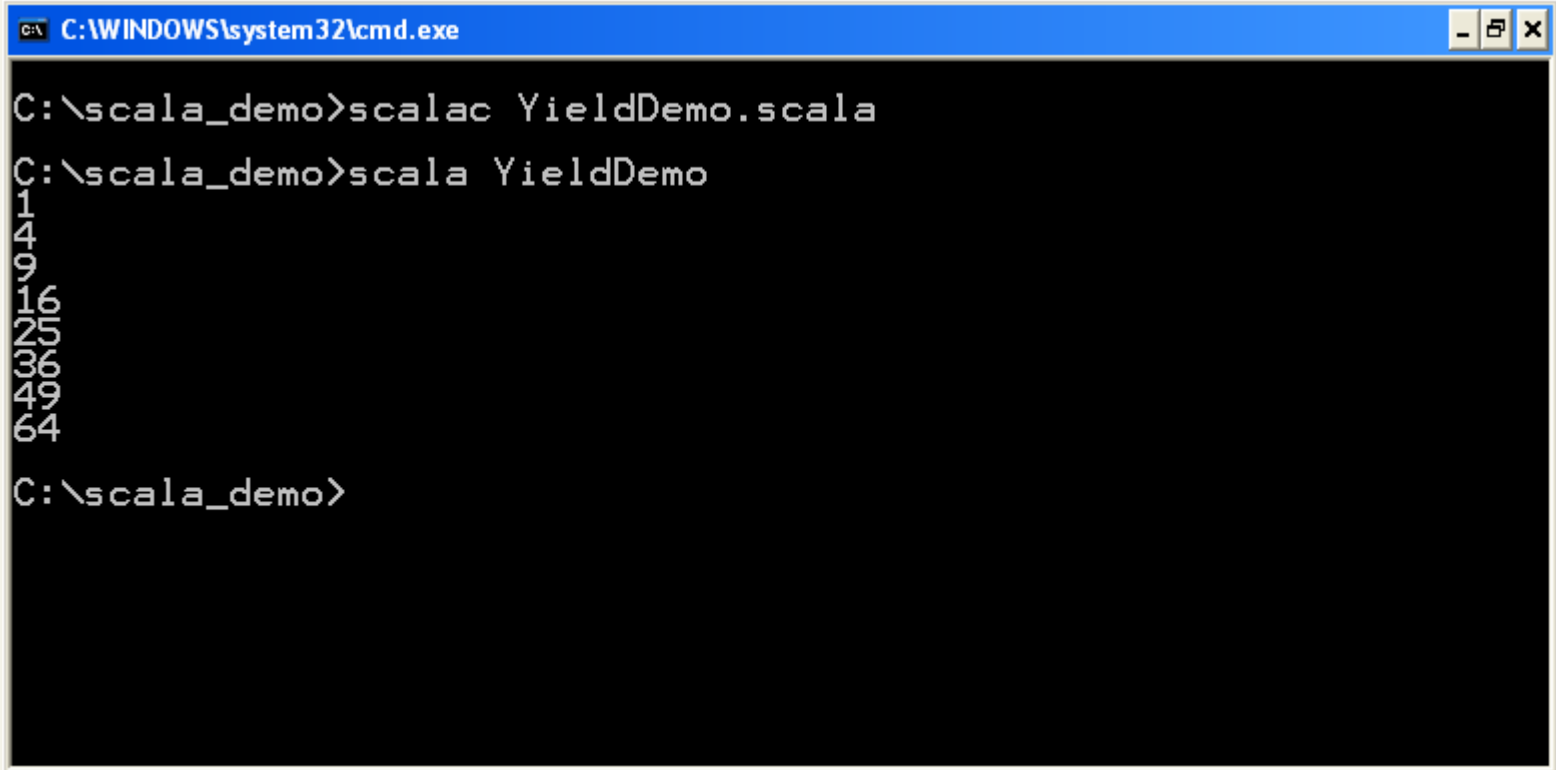
The following code sample creates a new collection based on a given one.

The for-yield Expression



```
YieldDemo.scala - Notepad
File Edit Format View Help
object YieldDemo
{
    def main(args: Array[String])
    {
        var numbers = Array(1,2,3,4,5,6,7,8)
        var sum: Int = 0
        var vec = for{number <- numbers}
                    yield number*number
        for(num<-vec)
            println(num)
    }
}
```

The `for-yield` Expression



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac YieldDemo.scala
C:\scala_demo>scala YieldDemo
1
4
9
16
25
36
49
64
C:\scala_demo>
```

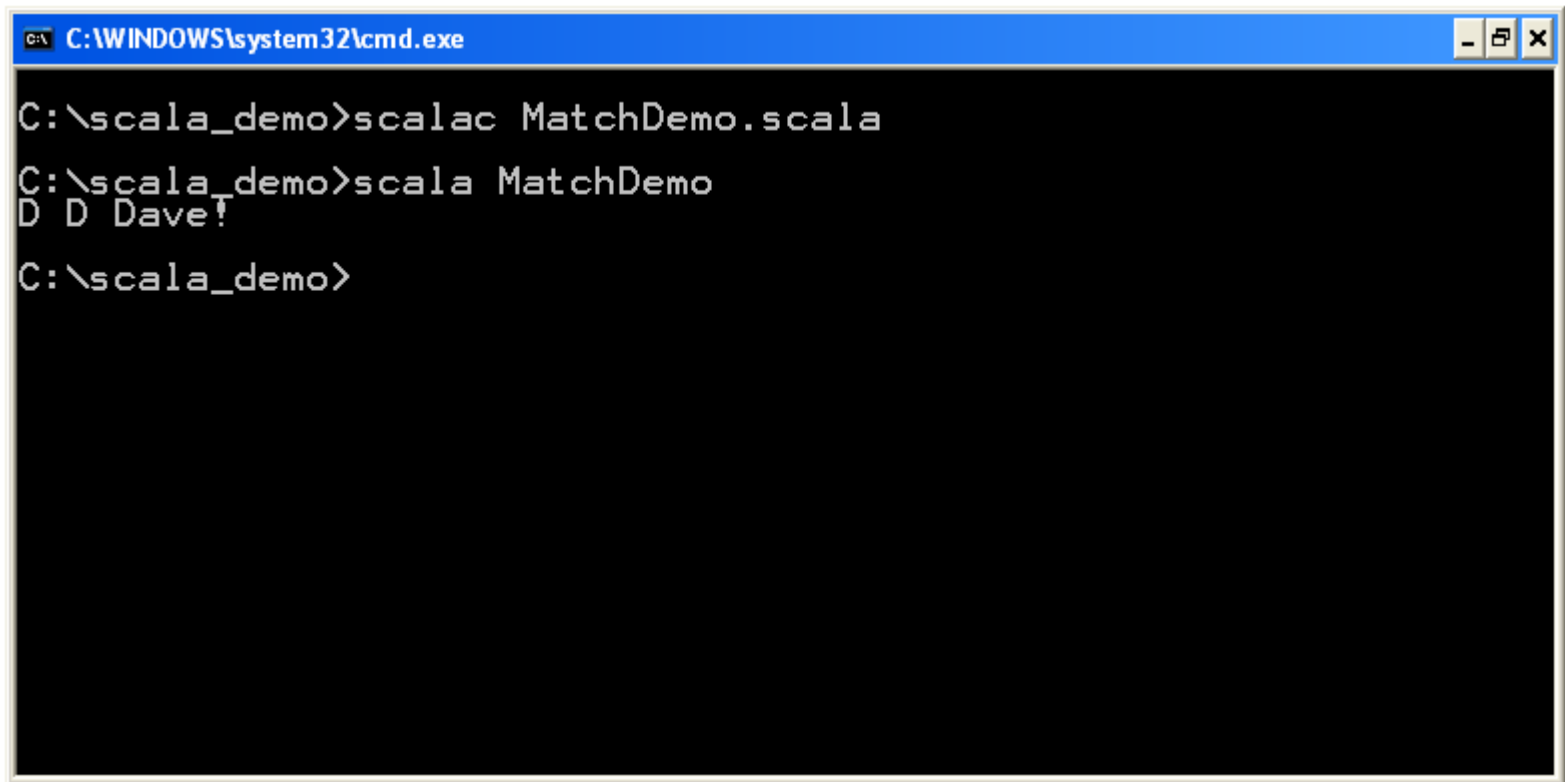

The `match` Expression

- ❖ The `match` expression is similar to the well known `switch` expression.

The match Expression

```
object MatchDemo
{
  def main(args: Array[String])
  {
    val name = "dave"
    name match
    {
      case "dave" => println("D D Dave!")
      case "java" => println("Janina J")
      case "fred" => println("frida fRedy")
    }
  }
}
```

The match Expression



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac MatchDemo.scala
C:\scala_demo>scala MatchDemo
D D Dave!
C:\scala_demo>
```

Break & Continue

- ❖ The Scala programming language doesn't support `break` and `continue`.

Variables Scope

- ❖ The Scala programming language supports variables scope the same way Java does.