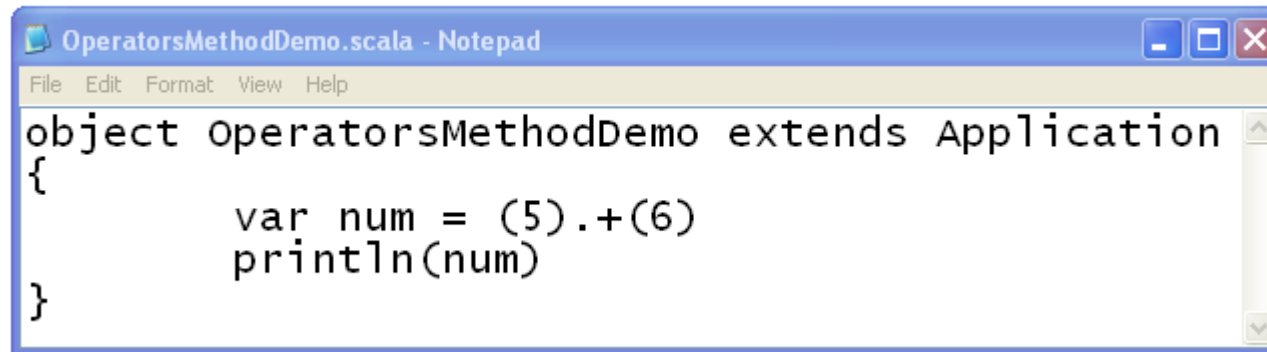


Operators

Operators & Methods

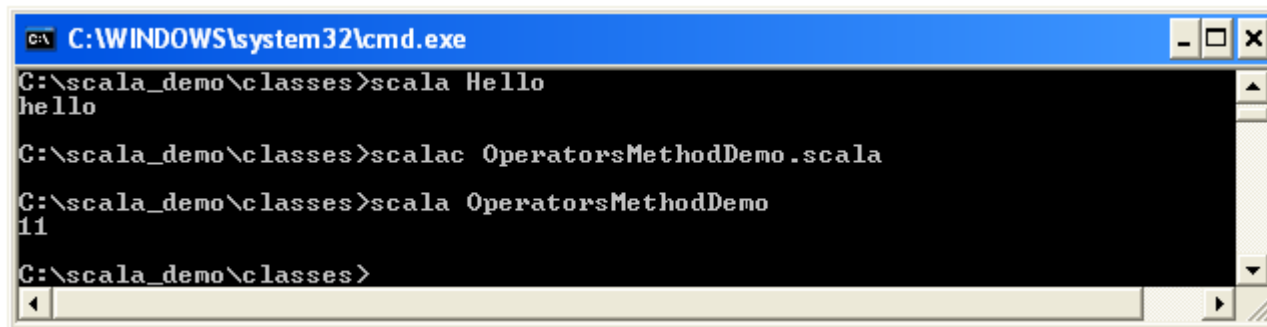
- ❖ The operators Scala supports are actually just another syntax for calling methods (e.g. When calling `1+2` that expression is actually translated into `(1) . + (2)).`

Operators & Methods



A Notepad window titled "OperatorsMethodDemo.scala - Notepad" with a menu bar (File, Edit, Format, View, Help). The code inside is:

```
object OperatorsMethodDemo extends Application
{
    var num = (5).+(6)
    println(num)
}
```



A Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe" showing the following commands and output:

```
C:\scala_demo\classes>scala Hello
hello

C:\scala_demo\classes>scalac OperatorsMethodDemo.scala

C:\scala_demo\classes>scala OperatorsMethodDemo
11

C:\scala_demo\classes>
```

Arithmetic Operators

❖ Scala supports the following common well known arithmetic operators:

+ - = * %

Relational Operators

- ❖ Scala supports the following common well known relational operators:

< <= > >= != ==

Logical Operators

- ❖ Scala supports the following common well known logical operators:

& & | | !

Bitwise Operators

- ❖ Scala supports the following common well known bitwise operators:

& | ^ ~ << >> >>>

Objects Comparison

- ❖ Comparing two objects using the `==` operator indirectly calls the `equals()` method on the operand on left passing over the operand on the right.



Operators Precedence

- ❖ The operators precedence is in accordance with the first letter of the method been called (e.g. operators that start with `*` precede operators that start with `+`).
- ❖ The assignment operators that end with the `=` character (e.g. `+=`, `-=`, `*=`) have the same priority the `=` operator has.

Operators Precedence

❖ The following table summarizes the operators precedences:

(all other special characters)

* / %

+ :

= <= >= == !=

< >

&

^

|

(all letters)

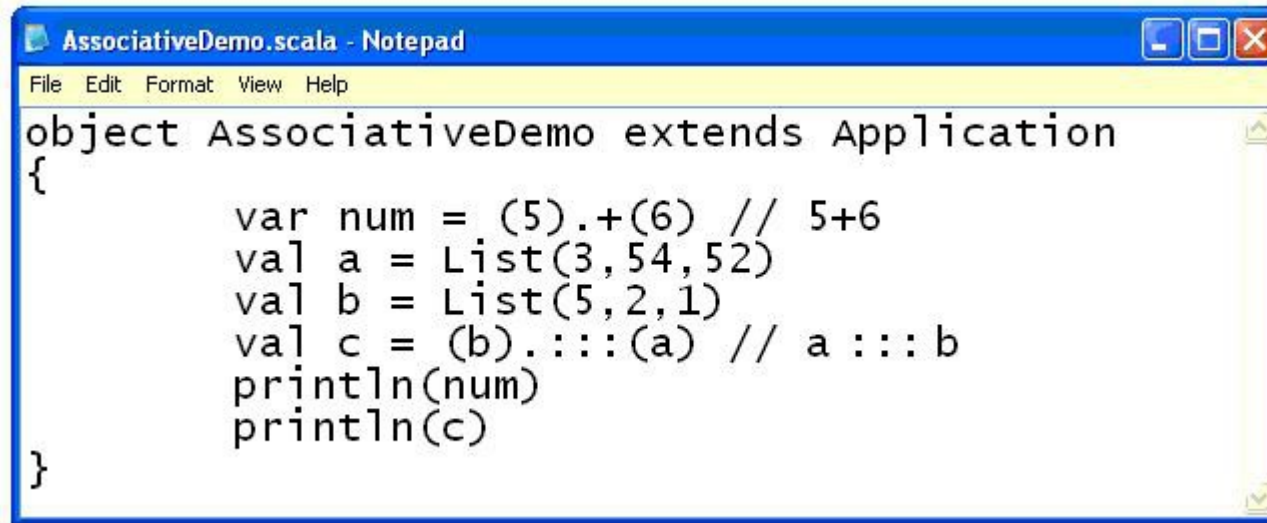
Operators Associative

- ❖ When operators with the same priority appear side by side the expression is evaluated in accordance with the operators associativity.
- ❖ In Scala, operators associativity is determined by the last character.

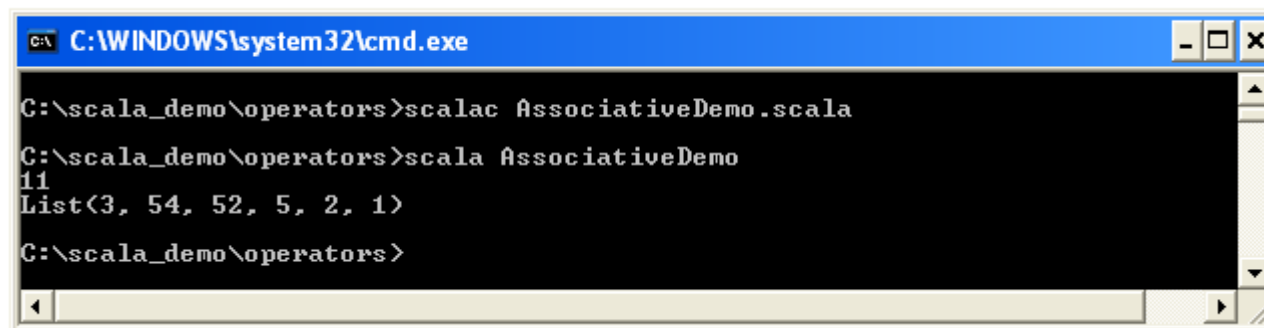
Operators Associative

- ❖ Operators that end with : will be translated into a method call on the right operand passing over the left operand as an argument. In all other cases, the method will be called on the left operand while having the right operand passed over as argument.
- ❖ If the last character isn't : then operators that have the same precedence will be evaluated left to right.

Operators Associative



```
AssociativeDemo.scala - Notepad
File Edit Format View Help
object AssociativeDemo extends Application
{
    var num = (5).+(6) // 5+6
    val a = List(3,54,52)
    val b = List(5,2,1)
    val c = (b).:::(a) // a ::: b
    println(num)
    println(c)
}
```



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo\operators>scalac AssociativeDemo.scala
C:\scala_demo\operators>scala AssociativeDemo
11
List(3, 54, 52, 5, 2, 1)
C:\scala_demo\operators>
```

Wrapper Classes

- ❖ For each one of the primitive type classes there is an available wrapper class that includes additional methods we can call on our basic type values.

Wrapper Classes

Basic Type

Rich Wrapper

Byte

`scala.runtime.RichByte`

Short

`scala.runtime.RichShort`

Int

`scala.runtime.RichInt`

char

`scala.runtime.RichChar`

String

`scala.runtime.RichString`

Float

`scala.runtime.RichFloat`

Double

`scala.runtime.RichDouble`

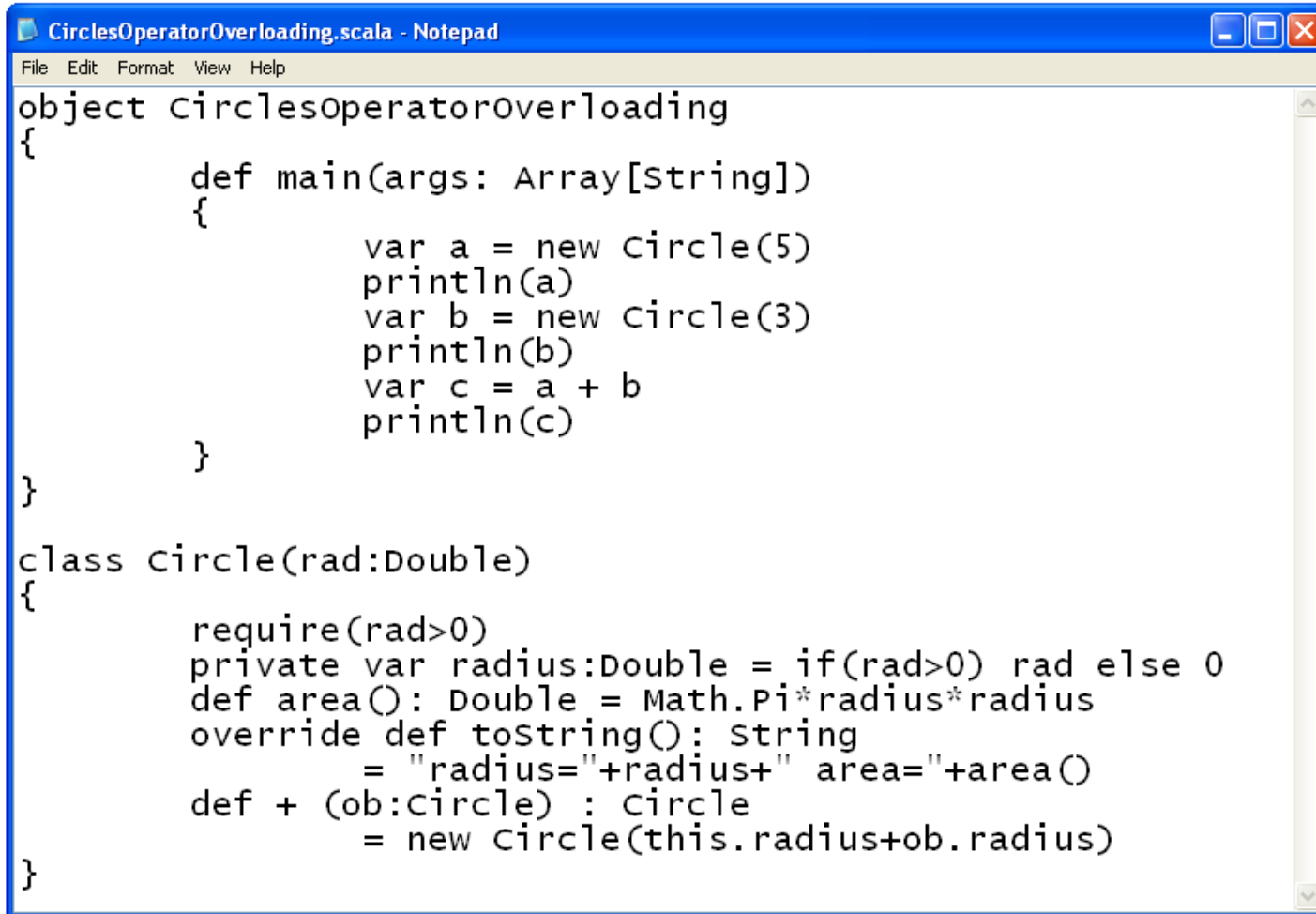
Boolean

`scala.runtime.RichBoolean`

Operators Overloading

- ❖ Scala allows us to overload operators in a class we define. Overloading an operator is done by defining a new method that its name is the operator we want to overload.

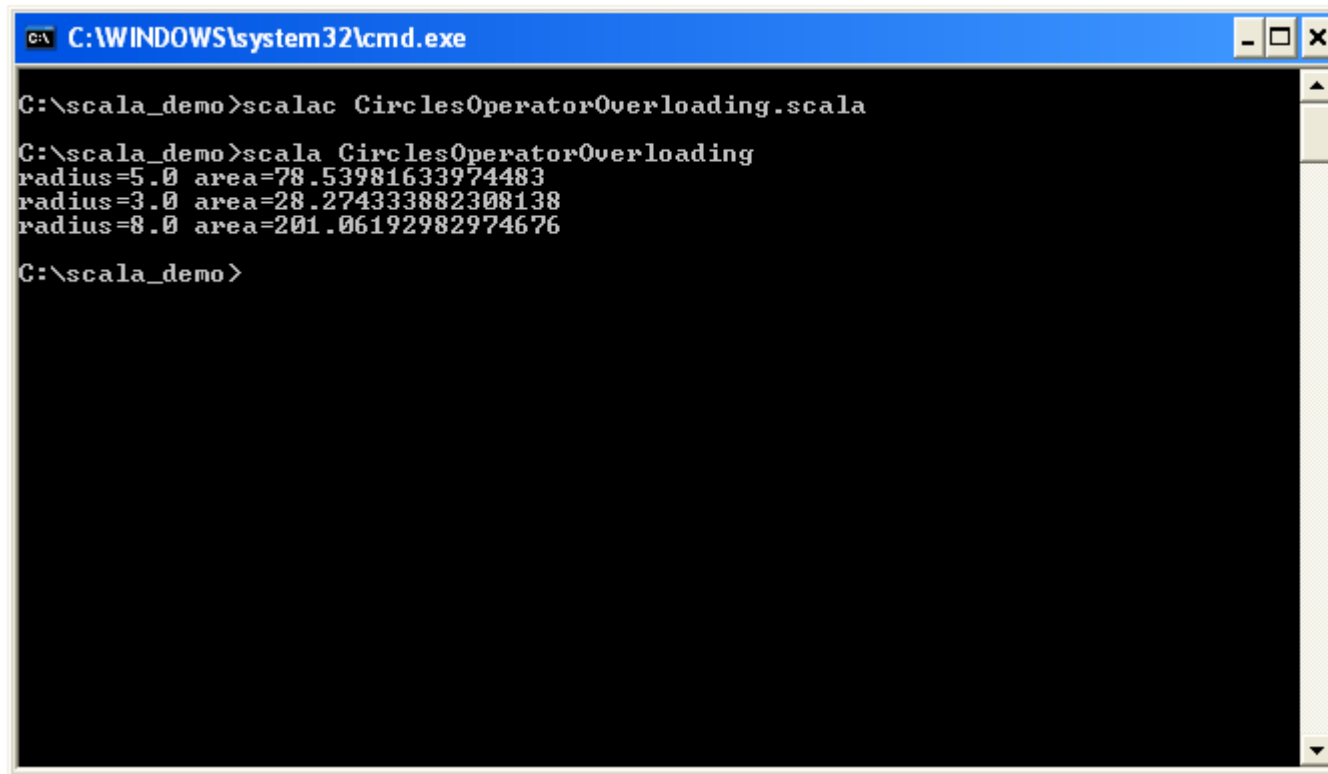
Operators Overloading



```
CirclesOperatorOverloading.scala - Notepad
File Edit Format View Help
object CirclesOperatorOverloading
{
    def main(args: Array[String])
    {
        var a = new Circle(5)
        println(a)
        var b = new Circle(3)
        println(b)
        var c = a + b
        println(c)
    }
}

class Circle(rad:Double)
{
    require(rad>0)
    private var radius:Double = if(rad>0) rad else 0
    def area(): Double = Math.Pi*radius*radius
    override def toString(): String
        = "radius="+radius+" area="+area()
    def + (ob:Circle) : Circle
        = new Circle(this.radius+ob.radius)
}
```

Operators Overloading



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

C:\scala_demo>scalac CirclesOperatorOverloading.scala

C:\scala_demo>scala CirclesOperatorOverloading
radius=5.0 area=78.53981633974483
radius=3.0 area=28.274333882308138
radius=8.0 area=201.06192982974676

C:\scala_demo>
```

The unary_ Operator

- ❖ When overloading an unary operator we should precede it with the `unary_` operator.

The unary_ Operator

```
package il.ac.hit.samples

object Program
{
  def main(args: Array[String])
  {
    val ob = new RationalNumber(1,3)
    val other = -ob
    println(other)
  }
}
```



The unary_ Operator

```
class RationalNumber(a:Int,b:Int)
{
  require (b!=0,"donominator cannot be 0")
  val numerator:Int = a / greatestCommonDivider(a,b)
  val denominator:Int = b / greatestCommonDivider(a,b)
  private def greatestCommonDivider(m:Int,n:Int):Int =
  {
    if(n==0) m else greatestCommonDivider(n,m%n)
  }
  override def toString():String =
  {
    numerator+"/"+denominator
  }
  def unary_- :RationalNumber =
    new RationalNumber(-numerator,denominator)
}
```

The unary_ Operator

