

Classes & Objects

Class Definition

- ❖ The class is kind of a template through which we can create new objects. We create new objects using the 'new' keyword.

...

```
class Rectangle
```

```
{
```

```
    ...
```

```
}
```

...

```
new Rectangle
```

...

Class Definition

- ❖ The fields we define within the class can be defined either using `val` or using `var`. Either way, these variables refer separately to each one of the objects instantiated from our class.
- ❖ Each value is an object. Including the primitive type values. Therefore, each field we define within the class becomes a variable within a specific object.

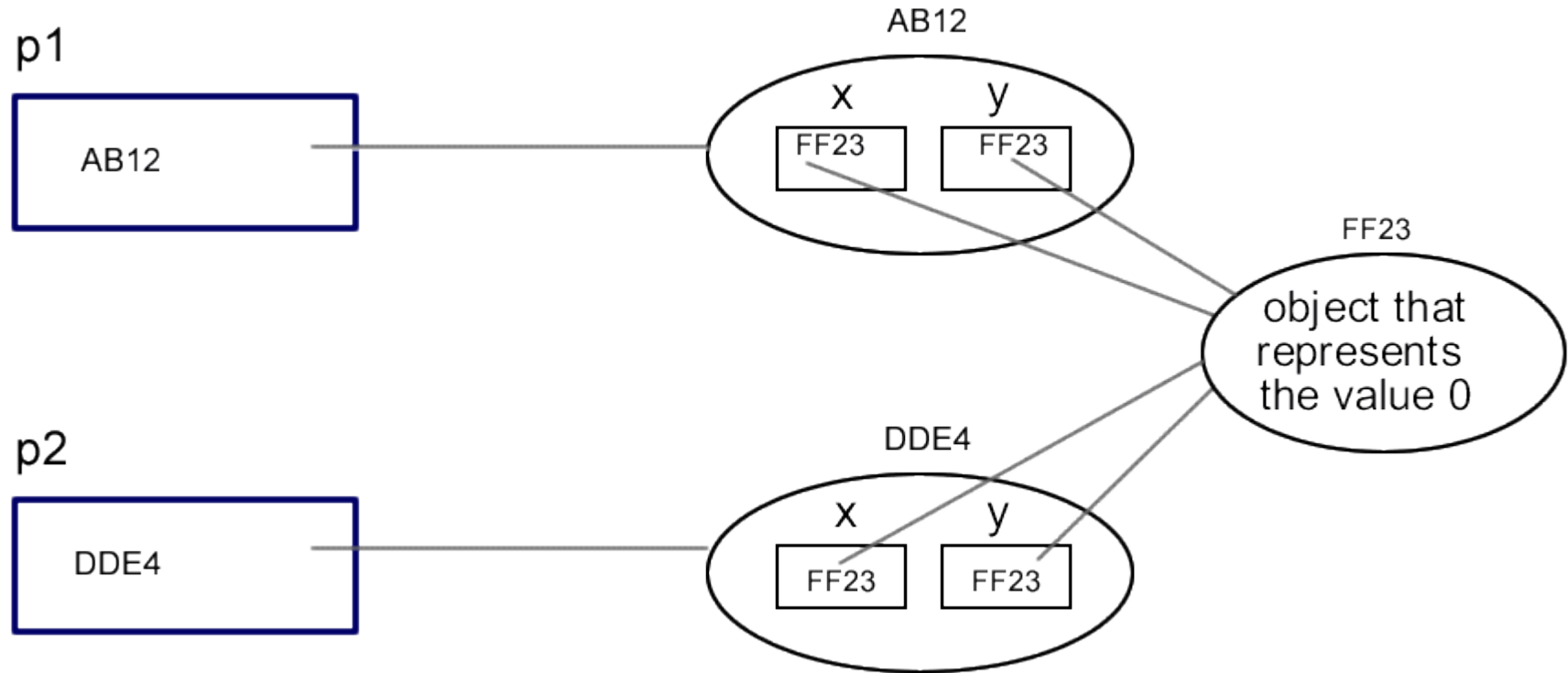
Class Definition

- ❖ Different objects that hold within that field the same primitive type value actually hold the same reference for the same object.

```
class Point
{
    var x:Int = 0
    var y:Int = 0
}

var p1 = new Point
var p2 = new Point
```

Class Definition



Class Definition

- ❖ When new values will be assigned to each one of the coordinates of each one of the two points we will get a new graph of objects reflecting that.

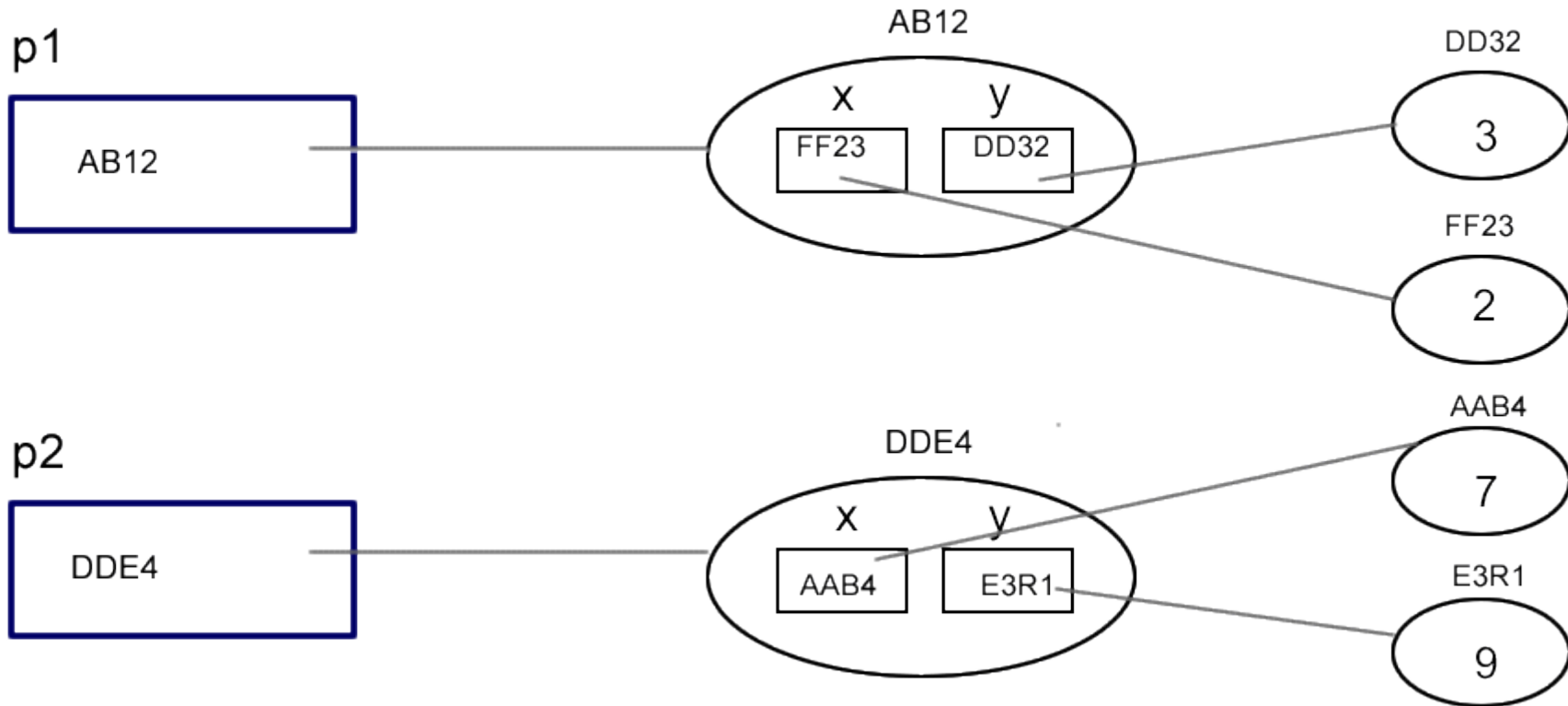
`p1.x = 2`

`p1.y = 3`

`p2.x = 7`

`p2.y = 9`

Class Definition



Visibility Rules

- ❖ The keywords that modify visibility appear at the beginning of declarations. We will find them before the `class` or the `trait` keywords for types, before the `var` or `val` keywords for fields, and before the `def` keyword for methods.
- ❖ When applying a visibility modifier for constructor we should place it after the type name and before the argument list.

...

```
class Student private (name: String) {...}
```

...

The `private` Access Modifier

- ❖ We can protect the values the instance variables hold by defining them with the `'private'` access modifier.

```
class Rectangle
{
    private var width:Double
    private var height:Double
    def setWidth(w:Double):Unit =
    {
        if(w>0) width = w
    }
    ...
}
```

The `private` Access Modifier

- ❖ Unlike Java, if an inner class has a private member the enclosing class cannot see it.

...

```
class Human
{
    class Brain
    {
        private var iq:Int;
        ...
    }
}
...
```

The `private` Access Modifier

- ❖ We can limit the visibility to the very same instance by writing `private[this]`. Another instance of the same class won't be able to access a member that was defined with the `private[this]` access modifier.

The `private` Access Modifier

...

```
package scopeA
```

```
{
```

```
class Box(private[this] val num: Int)
```

```
{
```

```
    def equalField(other: Box) = this.num == other.num
```

```
}
```

```
...
```

This code doesn't compile!

The `private` Access Modifier

- ❖ We can limit the visibility to specific type by writing `private [T]`, where `T` is the type.
- ❖ The accessibility won't be allowed neither from an inner type or from an outer one.

The `private` Access Modifier

...

```
package scopeA
```

```
{
```

```
  class Box(private[Box] val num: Int)
```

```
  {
```

```
    def equalField(other: Box) = this.num == other.num
```

```
    class Fly
```

```
    {
```

```
      def doSomething = num //ERROR
```

```
      ...
```

```
    }
```

```
  }
```

```
}
```

...

The `private` Access Modifier

- ❖ We can limit the visibility to specific package by writing `private[scope]`, where `scope` is the name of the package.
- ❖ The accessibility won't be allowed neither from an inner package, from an outer one or from a none-related one.

The `private` Access Modifier

```
...
package pack1
{
    package pack11
    {
        private [pack11] class A
    }
}
package pack2
{
    class C extends pack11.A // ERROR
}
...
```


The `protected` Access Modifier

- ❖ Protected members are visible to the defining type, to the derived types and to the nested ones.
- ❖ Protected types are visible only within the same package and within sub packages.
- ❖ Similarly to the private access modifier, we can limit the protected accessibility to specific `package`, `type` and `this`.

The `public` Access Modifier

- ❖ This is the default access modifier. When we don't specify a specific other access modifier this is the one that takes place.
- ❖ Public members and types are visible everywhere, across all boundaries.

Method Parameters

- ❖ The parameters we define in our methods are 'val' by default.

```
class Rectangle
{
    private var width:Double
    private var height:Double
    def setWidth(w:Double):Unit =
    {
        if(w>0) width = w
    }
    ...
}
```

Shorter Syntax

- ❖ When a method spans over one statement only we can take out the curly brackets.

```
class Rectangle
{
    private var width:Double
    private var height:Double
    def setWidth(w:Double):Unit = if(w>0) width = w
    def setHeight(h:Double):Unit = if(h>0) height = h
    def getWidth():Double = width
    def getHeight():Double = height
}
```

The Semicolon Inference

- ❖ When the statement spans over one line only it isn't necessary to end it with a semicolon (';').
- ❖ When writing multiple statement in the same line we must place a semicolon between each one of them.

```
val str = 'abc'; println(str);
```

Static Members

- ❖ Classes cannot have static members. We cannot define static fields and we cannot define static methods.

The `this` Keyword

- ❖ The `this` keyword refers to the object instance on which the currently executing method was invoked.

...

```
def area(): Double =  
{  
    Math.Pi*this.radius*this.radius  
}
```

...

Defining Constructors

- ❖ Defining the class primary constructor is done together with the class declaration at the same line of code. The compiler takes the parameters we specify in the class declaration (also known as the class parameters) and creates a primary constructor.

```
class Circle(rad:Double)
{
    ...
}
```


Defining Constructors

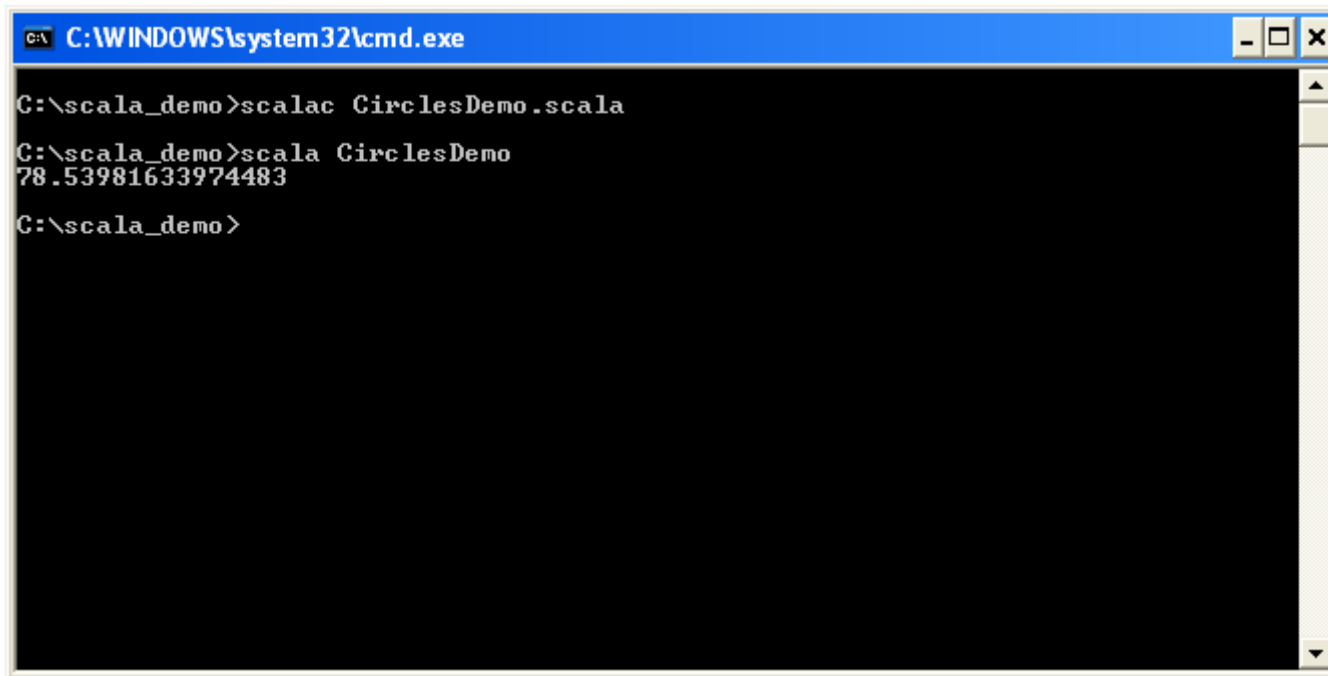
- ❖ The class parameters can be used directly in the body of the class. This helps us writing shorter code.

Defining Constructors

```
CirclesDemo.scala - Notepad
File Edit Format View Help
object CirclesDemo
{
    def main(args: Array[String])
    {
        var ob = new Circle(5)
        println(ob.area())
    }
}

class Circle(rad:Double)
{
    private var radius:Double = if(rad>0) rad else 0
    def area(): Double = Math.Pi*radius*radius
}
```

Defining Constructors



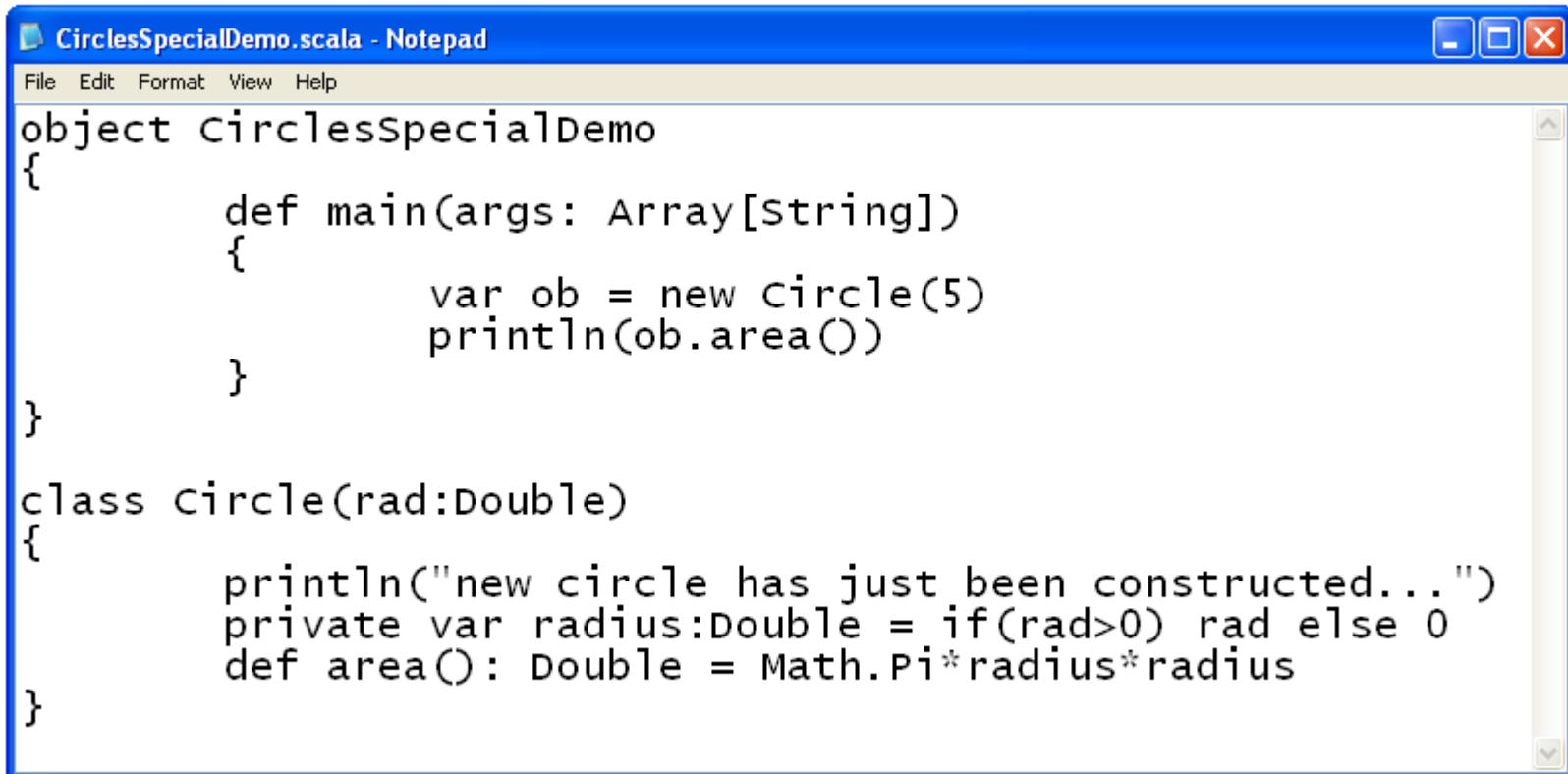
```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac CirclesDemo.scala
C:\scala_demo>scala CirclesDemo
78.53981633974483
C:\scala_demo>
```

Defining Constructors

- ❖ The Scala compiler will compile any code we place within the class body and which isn't part of a field or a method definition into the scope of the primary constructor.

```
class Circle(rad:Double)
{
  print ("new circle has just been constructed...")
  private var radius:Double = if(rad>0) rad else 0
  def area(): Double = Math.Pi*radius*radius
}
```

Defining Constructors

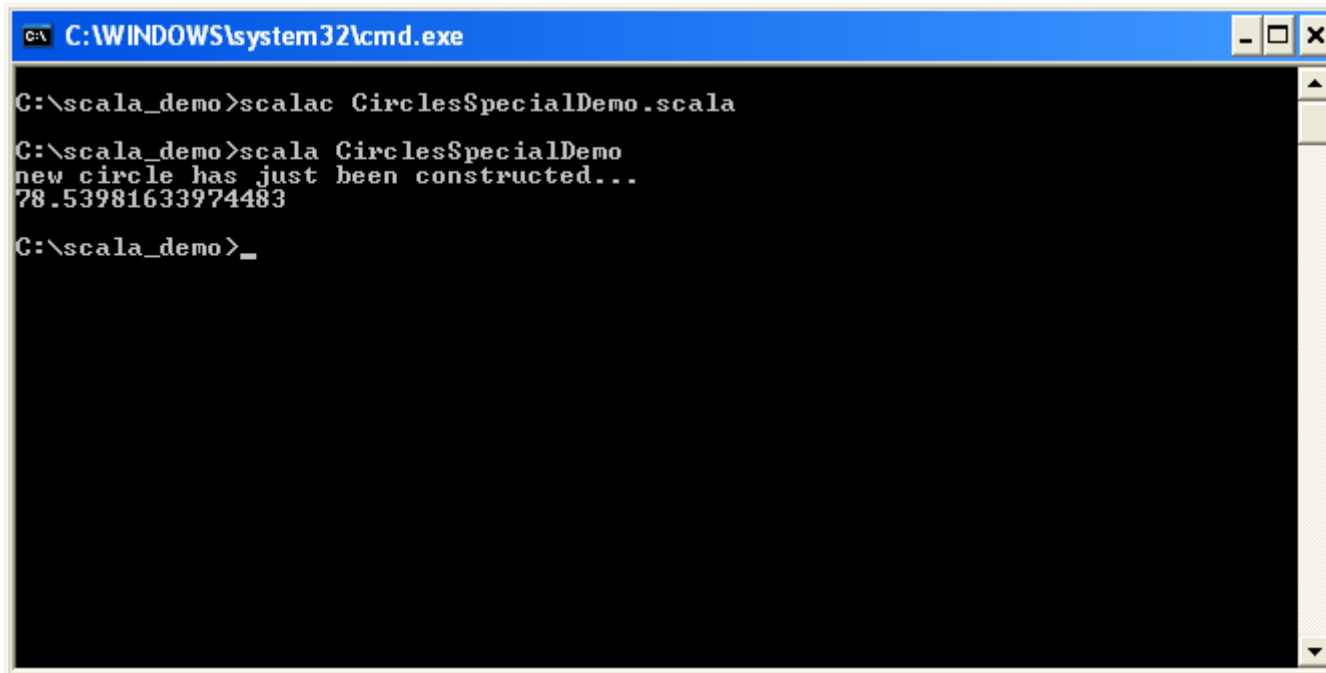


```
CirclesSpecialDemo.scala - Notepad
File Edit Format View Help
object CirclesSpecialDemo
{
    def main(args: Array[String])
    {
        var ob = new Circle(5)
        println(ob.area())
    }
}

class Circle(rad:Double)
{
    println("new circle has just been constructed...")
    private var radius:Double = if(rad>0) rad else 0
    def area(): Double = Math.Pi*radius*radius
}

```

Defining Constructors



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac CirclesSpecialDemo.scala
C:\scala_demo>scala CirclesSpecialDemo
new circle has just been constructed...
78.53981633974483
C:\scala_demo>
```

Defining Constructors

- ❖ Using `this` we can define more than one constructor.
- ❖ Using `this` we can call a specific constructor from within another one. That call should be the first statement.

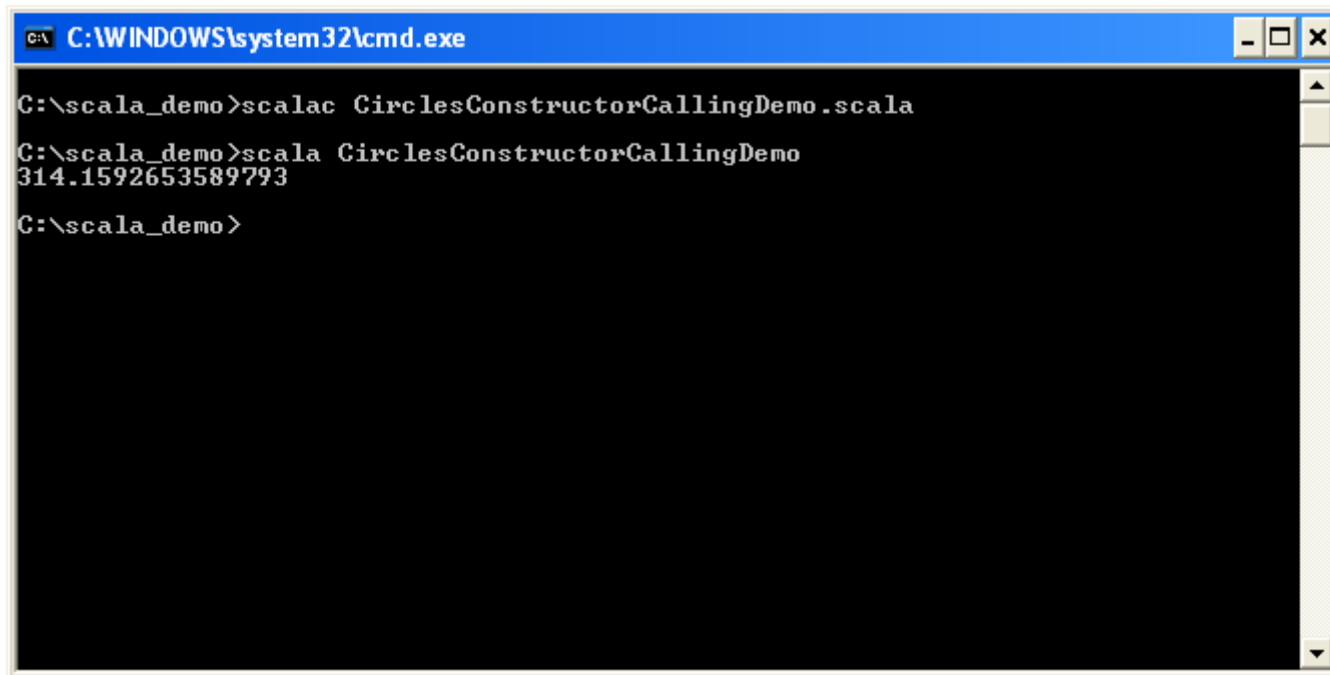
```
class Circle(rad:Double)
{
    private var radius:Double = if(rad>0) rad else 0
    def this() = this(10)
    def area(): Double = Math.Pi*radius*radius
}
```

Defining Constructors

```
CirclesConstructorCallingDemo.scala - Notepad
File Edit Format View Help
object CirclesConstructorCallingDemo
{
    def main(args: Array[String])
    {
        var ob = new circle()
        println(ob.area())
    }
}

class circle(rad:Double)
{
    private var radius:Double = if(rad>0) rad else 0
    def this() = this(10)
    def area(): Double = Math.Pi*radius*radius
}
```


Defining Constructors



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac CirclesConstructorCallingDemo.scala
C:\scala_demo>scala CirclesConstructorCallingDemo
314.1592653589793
C:\scala_demo>
```

Singleton Objects

- ❖ When defining a singleton object, instead of using `class` we use `object`.
- ❖ If we define a class with the same name in the same source file, the defined class is called the companion class of the singleton object. In this case, the singleton object and the companion class can access each other private members.

Singleton Objects

- ❖ We can use a companion class for developing a factory, as in the following code sample.

```
package com.lifemichael.samples

class Something(foo: String)

object Something {
  def apply(foo: String) = new Something(foo)
}

object Demo {

  def main(args:Array[String]):Unit = {
    var ob = Something("gaga")
    println("hello!")
  }
}
```

Singleton Objects

- ❖ A Singleton object can extend a super class and it can mix in traits.
- ❖ The main difference between singleton objects and classes is that we cannot instantiate a singleton object using the 'new' keyword.
- ❖ When defining a singleton object without sharing the name with a companion class the object is known as a standalone object.

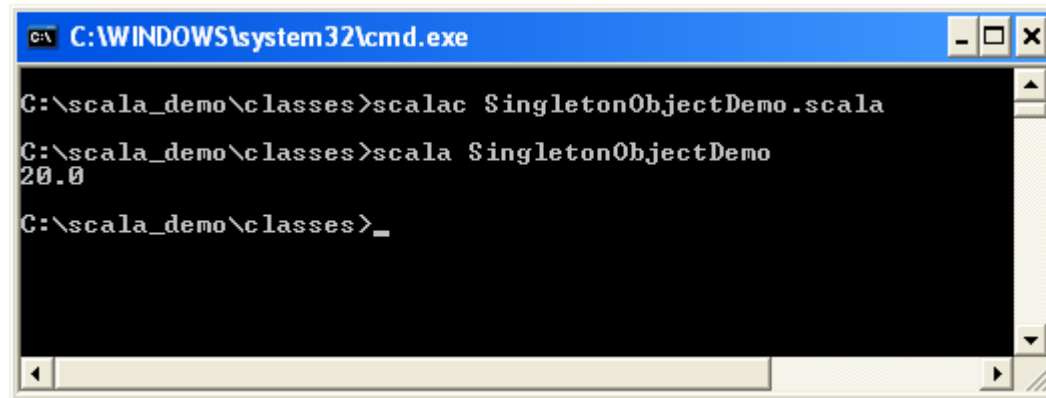
Singleton Objects

```
SingletonObjectDemo.scala - Notepad
File Edit Format View Help
class Account
{
    private var balance:Double = 0
    private def deposit(sum:Double) = balance+=sum
    private def withdraw(sum:Double) = balance-=sum
    private def details() = println(balance)
}

object Account
{
    var ob = new Account()
    def depositMoney(sum:Double) = ob.deposit(sum)
    def withdrawMoney(sum:Double) = ob.withdraw(sum)
    def printDetails() = ob.details()
}

object SingletonObjectDemo
{
    def main(args:Array[String])
    {
        Account.depositMoney(100)
        Account.withdrawMoney(80)
        Account.printDetails()
    }
}
```

Singleton Objects

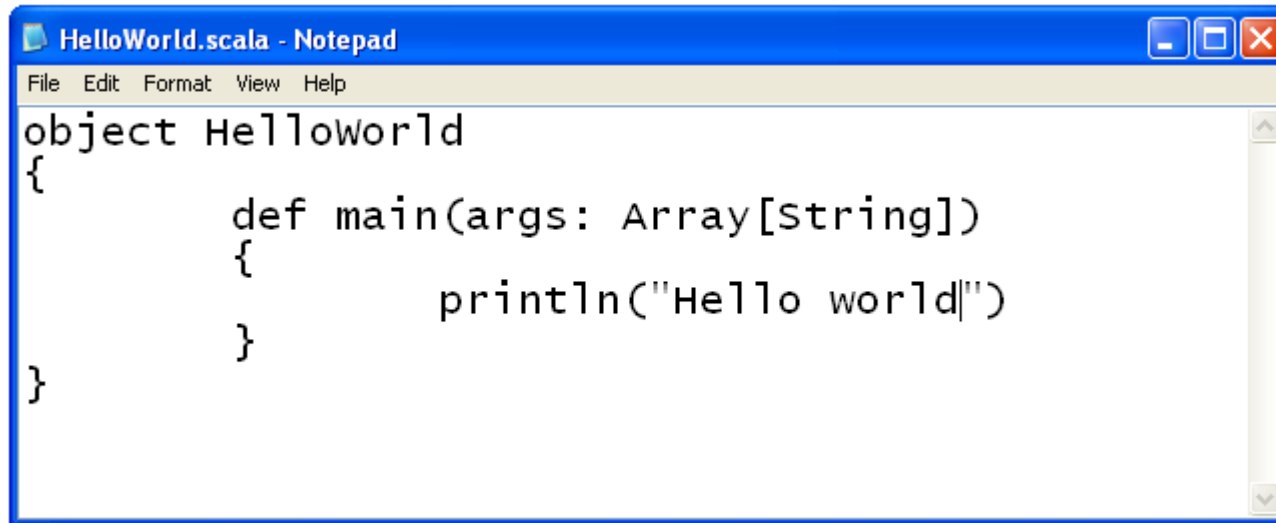


```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo\classes>scalac SingletonObjectDemo.scala
C:\scala_demo\classes>scala SingletonObjectDemo
20.0
C:\scala_demo\classes>_
```

Standalone Application

- ❖ In order to develop a stand alone application we should define a singleton object with the method `main`. The `main` method should be defined with one parameter of type `Array[String]`. In addition, its returned value should be of type `Unit`. The `main` method is the application entry point.

Standalone Application

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

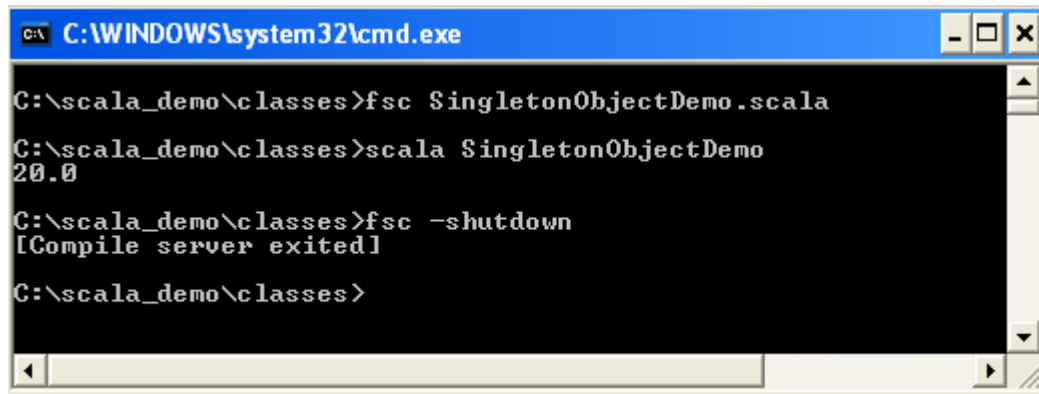
```
object HelloWorld
{
    def main(args: Array[String])
    {
        println("Hello world")
    }
}
```


The `fsc` Compiler

- ❖ Using the '`fsc`' compiler (instead of '`scalac`') can dramatically shortcut the development time.
- ❖ The first time we call '`fsc`' a small local server daemon attached to a port on our computer will start running.
- ❖ The second time we call '`fsc`' the small local server daemon will be already up and running. Passing over the names of the files we want to compile the compilation will be dramatically faster.

The `fsc` Compiler

- ❖ Calling '`fsc -shutdown`' will shut down the small daemon server.



```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo\classes>fsc SingletonObjectDemo.scala
C:\scala_demo\classes>scala SingletonObjectDemo
20.0
C:\scala_demo\classes>fsc -shutdown
[Compile server exited]
C:\scala_demo\classes>
```

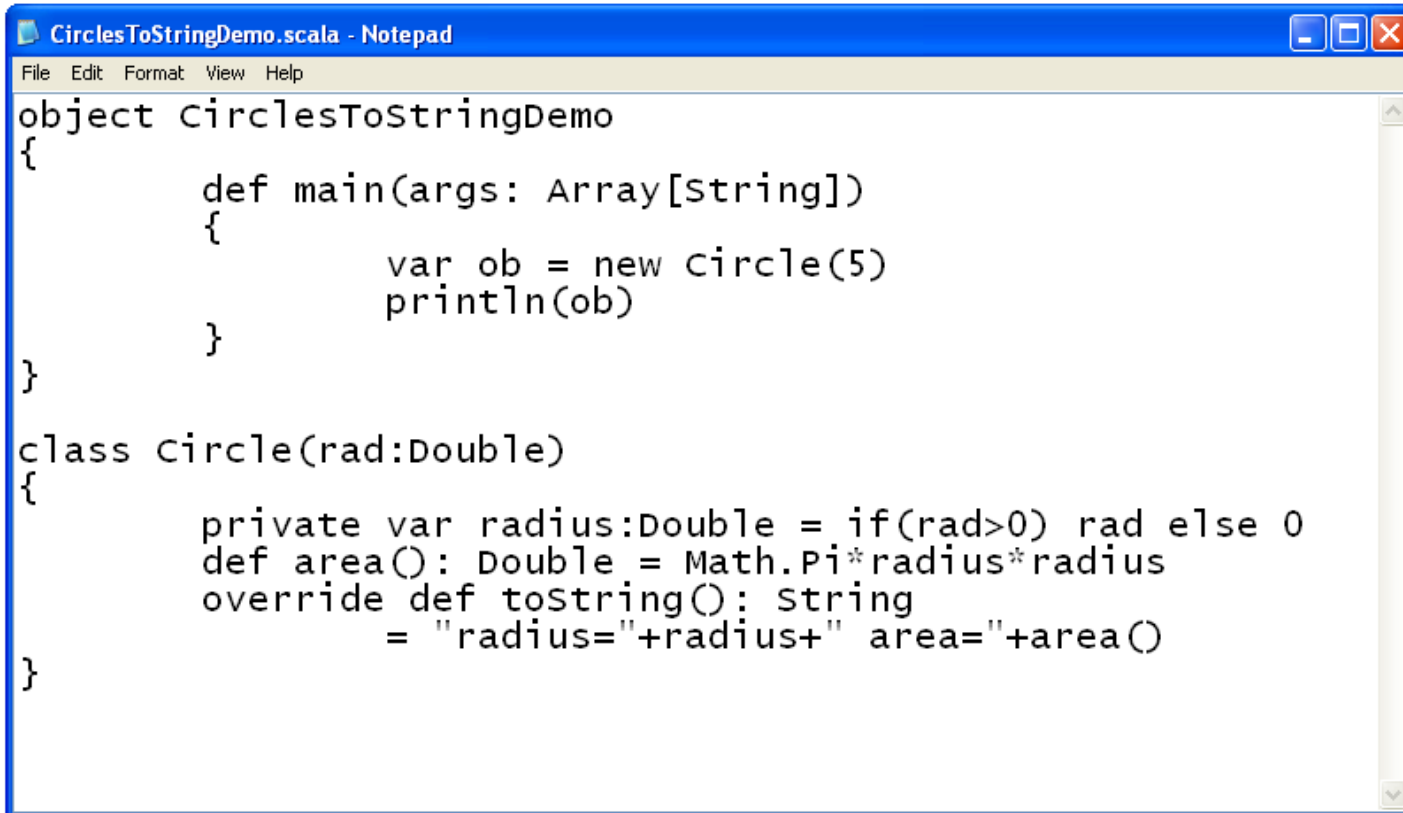
The `java.lang.Object` Class

- ❖ All classes extend `java.lang.Object`. We can call each one of the methods defined in `java.lang.Object` on every object of every type.

Overriding Methods

- ❖ When overriding a method we should use the `override` modifier.
- ❖ The following code sample shows how to override the `toString()` method.

Overriding Methods

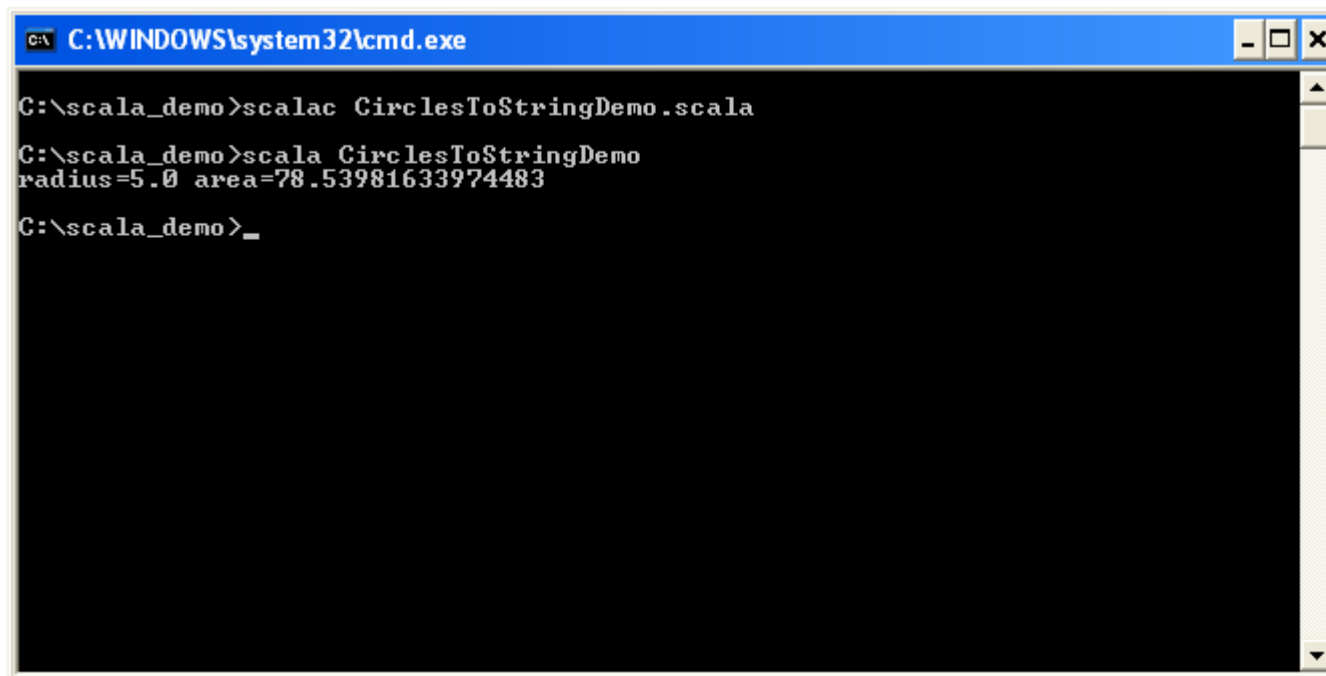


```
CirclesToStringDemo.scala - Notepad
File Edit Format View Help
object CirclesToStringDemo
{
    def main(args: Array[String])
    {
        var ob = new Circle(5)
        println(ob)
    }
}

class Circle(rad:Double)
{
    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()
}

```

Overriding Methods



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

C:\scala_demo>scalac CirclesToStringDemo.scala
C:\scala_demo>scala CirclesToStringDemo
radius=5.0 area=78.53981633974483
C:\scala_demo>_
```

The `require` Function

- ❖ Using this function we can specify a requirement for arguments passed over to the function.
- ❖ We can use it in order to enforce a precondition on the caller of the function. If the condition is not met then an `IllegalArgumentException` will be thrown.

The `require` Function

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

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()
}
```


The require Function

```
C:\WINDOWS\system32\cmd.exe
C:\scala_demo>scalac PreconditionCirclesDemo.scala
C:\scala_demo>scala PreconditionCirclesDemo
no such file: PreconditionCirclesDemo
C:\scala_demo>scalac PreconditionCirclesDemo.scala
C:\scala_demo>scala PreconditionCirclesDemo
radius=5.0 area=78.53981633974483
java.lang.IllegalArgumentException: requirement failed
    at scala.Predef$.require(Predef.scala:107)
    at Circle.<init><PreconditionCirclesDemo.scala:14>
    at PreconditionCirclesDemo$.main(PreconditionCirclesDemo.scala:7)
    at PreconditionCirclesDemo.main(PreconditionCirclesDemo.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at scala.tools.nsc.ObjectRunner$$anonfun$run$1.apply(ObjectRunner.scala:
75)
    at scala.tools.nsc.ObjectRunner$.withContextClassLoader(ObjectRunner.sca
la:49)
    at scala.tools.nsc.ObjectRunner$.run(ObjectRunner.scala:74)
    at scala.tools.nsc.MainGenericRunner$.main(MainGenericRunner.scala:154)
    at scala.tools.nsc.MainGenericRunner.main(MainGenericRunner.scala)
C:\scala_demo>
```

Method Overloading

- ❖ Scala supports methods overloading. We can define the same method in several different version. Each version should differ either in the number of parameters or their types.

Anonymous Inner Class

- ❖ The Scala programming language allows us to define anonymous inner classes.
- ❖ The syntax is very similar to the syntax we all know in Java.

Anonymous Inner Class

```
object HelloSample
{
  def main(args:Array[String]):Unit =
  {
    val ob = new MyStack[Int](0)
    {
      def data:Nothing = throw new Exception("empty stack");
    }
  }
}

abstract class MyStack[T](size:Int)
{
  def data:T;
}
```

