Inheritance

Introduction

- Inheritance is the well known relationship when having one class that extends the other.

Abstract Class

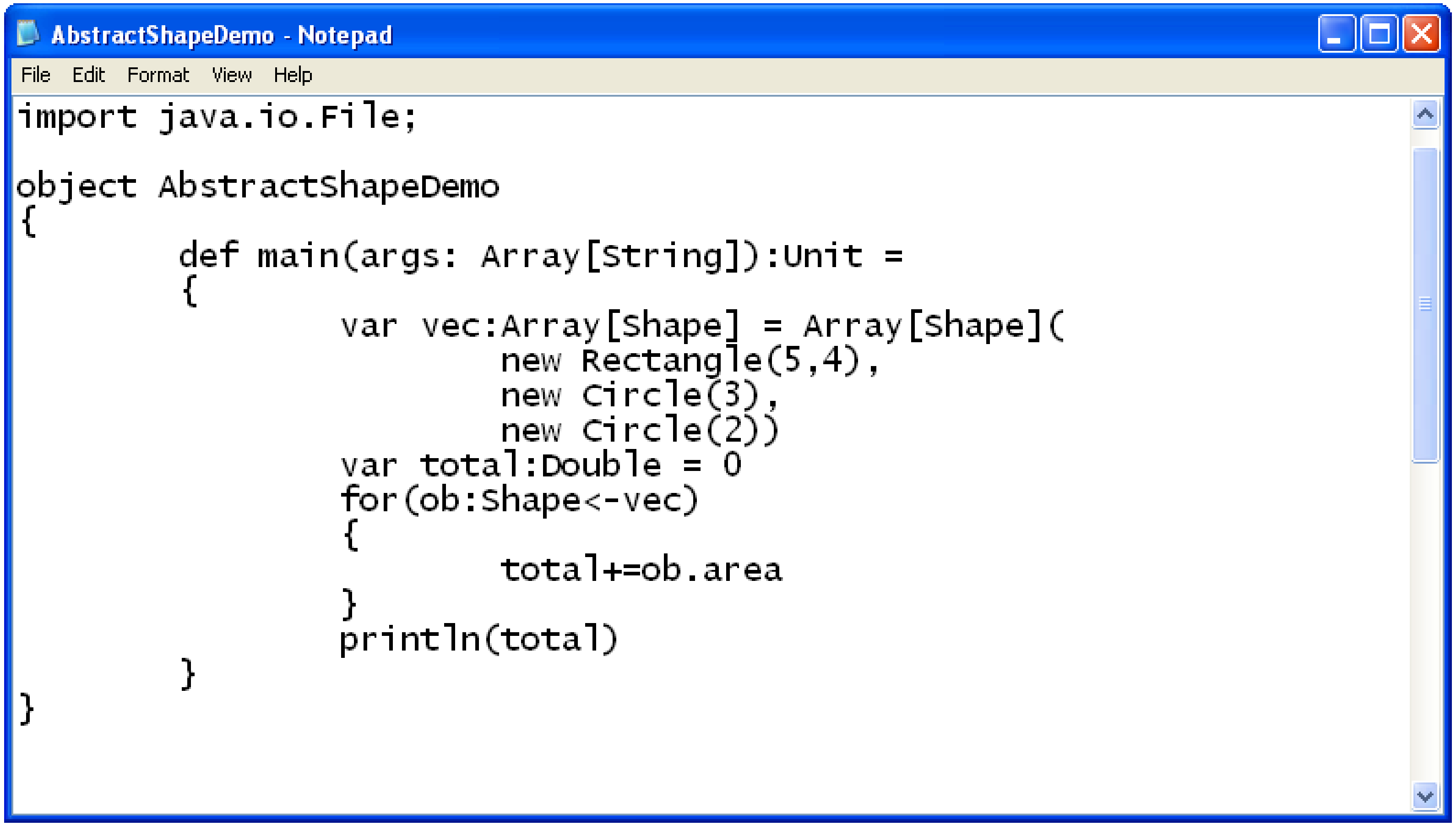
- We declare an abstract class by using the `abstract` keyword in the class declaration.
- We declare an abstract function by declaring it without a body.

```
...
abstract class Shape
{
    def area: Double
}
...
```

Abstract Class

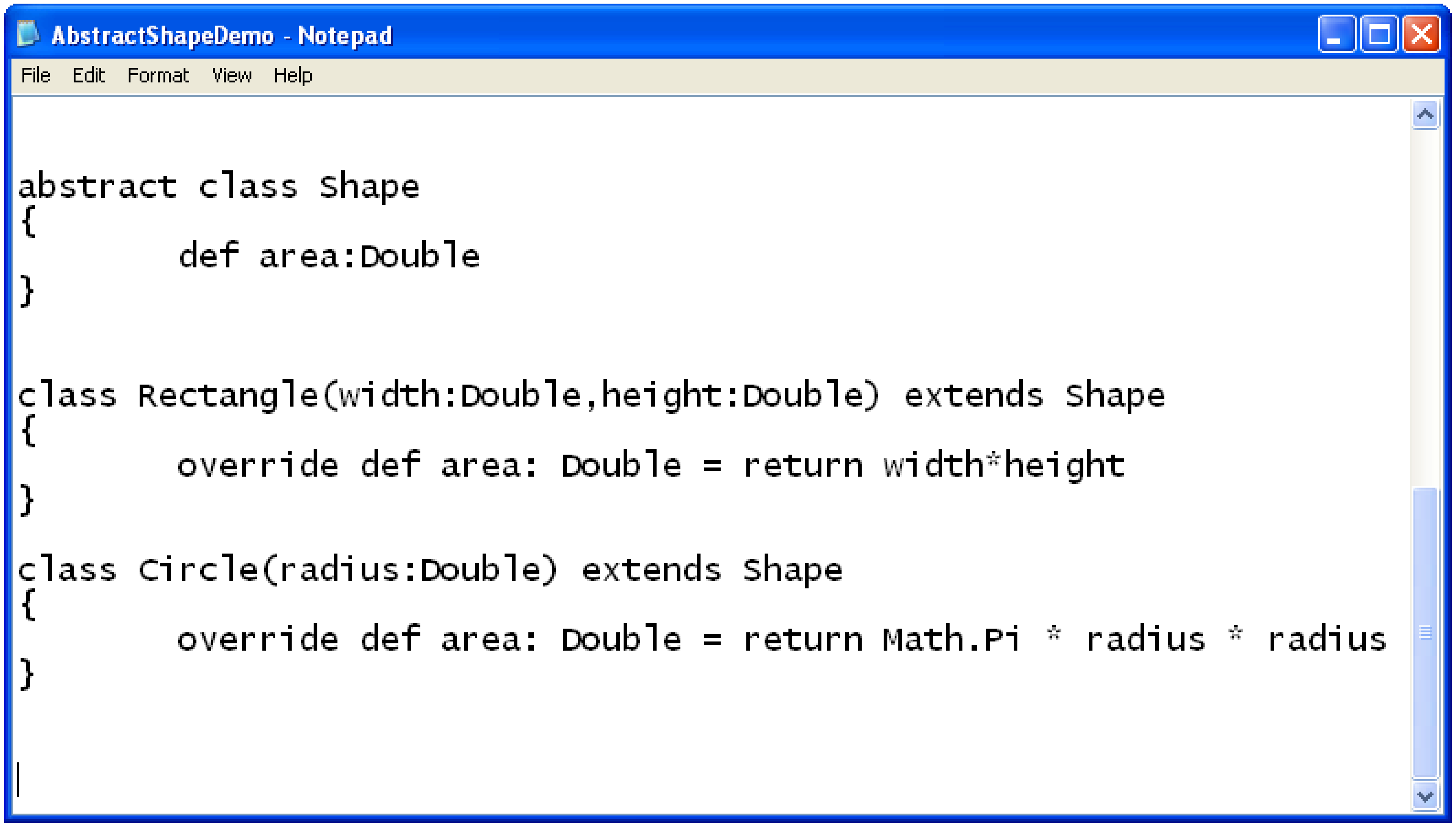
- Declaring a class with the abstract modifier indicates that the class might have abstract members and therefore we cannot instantiate it.
- Methods without implementation are abstract methods.
- Methods that do have a body are considered as concrete ones.
- Classes that extend an abstract class should include the definition for its abstract methods.

Abstract Class



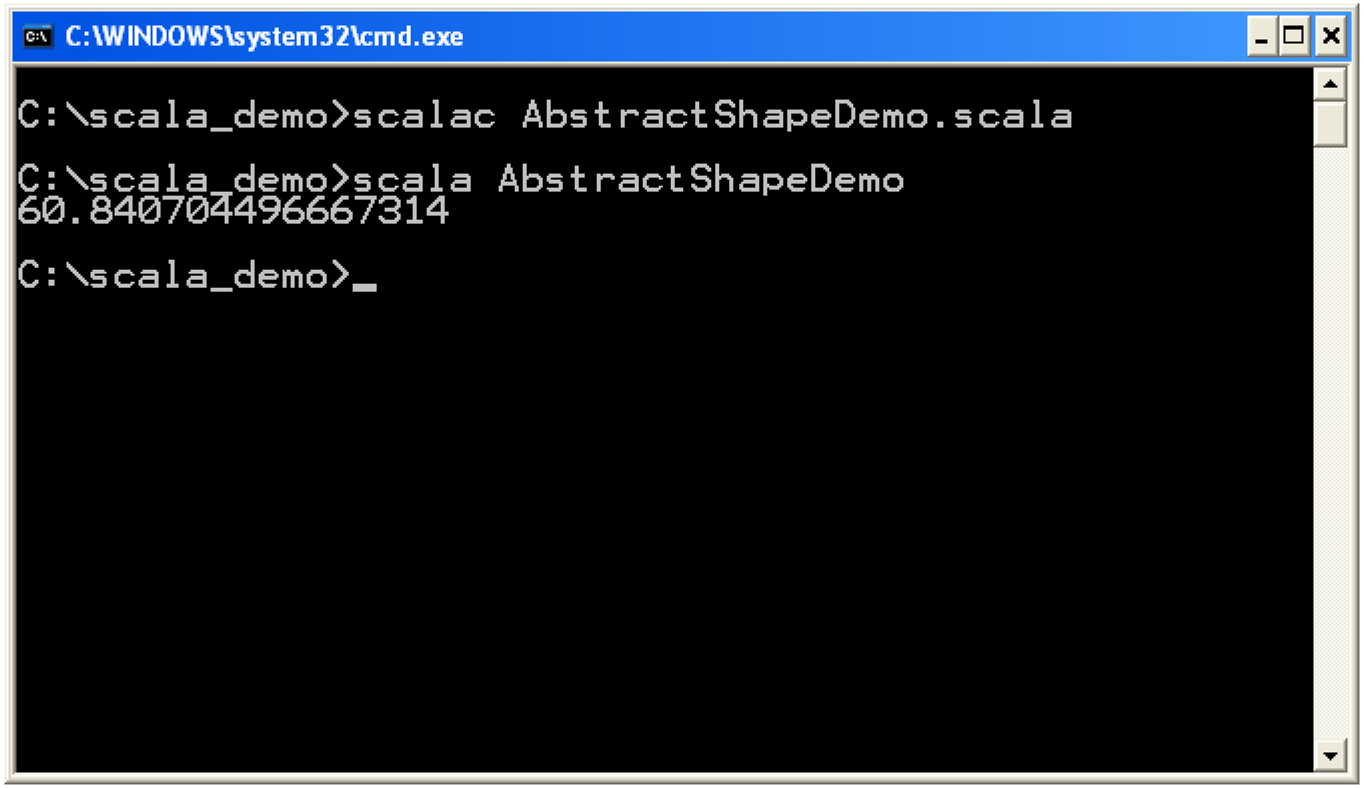
```
import java.io.File;

object AbstractShapeDemo
{
        def main(args: Array[String]):Unit =
        {
                var vec:Array[Shape] = Array[Shape](
                        new Rectangle(5,4),
                        new Circle(3),
                        new Circle(2))
                var total:Double = 0
                for(ob:Shape<-vec)
                {
                        total+=ob.area
                }
                println(total)
        }
}
```

Abstract Class



```
abstract class Shape
{
        def area:Double
}


class Rectangle(width:Double,height:Double) extends Shape
{
        override def area: Double = return width*height
}

class Circle(radius:Double) extends Shape
{
        override def area: Double = return Math.Pi * radius * radius
}
```

Abstract Class



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

C:\scala_demo>scala AbstractShapeDemo
60.840704496667314

C:\scala_demo>_
```

Parameter-less Methods

- When defining a method without parameters we can avoid the parentheses.

```
class Rectangle(width:Double,height:Double)
{
    def getWidth: Double = width
    def getHeight: Double = height
    def area: Double = width * height
}
```

Extending Classes

- We define one class as one that extends another using the `extends` keyword.

```
class Rectangle(width:Double,height:Double) extends Shape
{
    def getWidth: Double = width
    def getHeight: Double = height
}
```

Inheritance Meaning

- When having one class that extends another it means that all members of the base class are also members of the subclass.
- The private members exist in our new class as well. However, their accessibility is not direct.

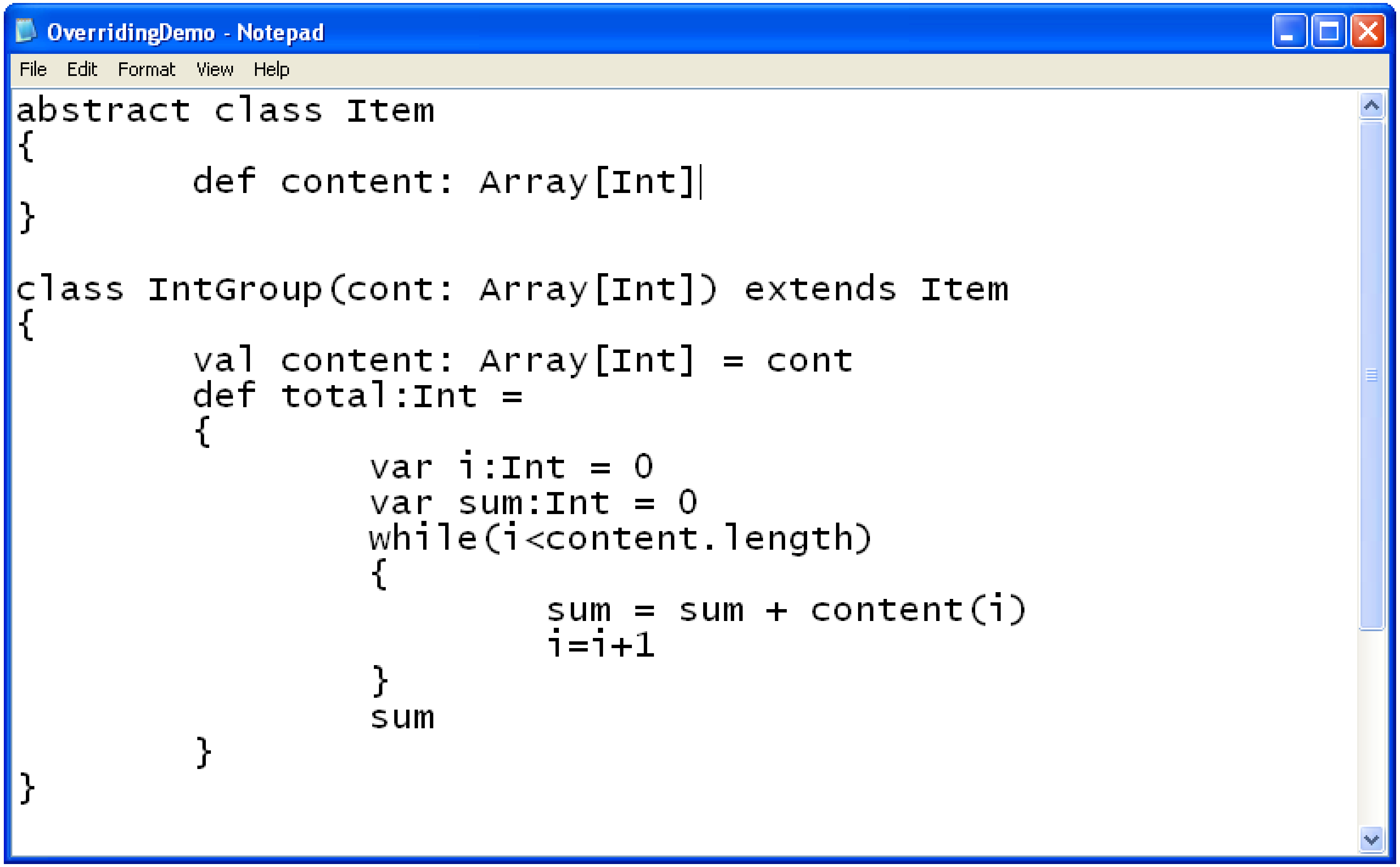
Inheritance Meaning

- When a member our class inherits is already defined in our class we can say that our class definition for that member either implements the inherited one (when the inherited one is abstract) or overrides it (when the inherited one is concrete).

Overriding Methods & Fields

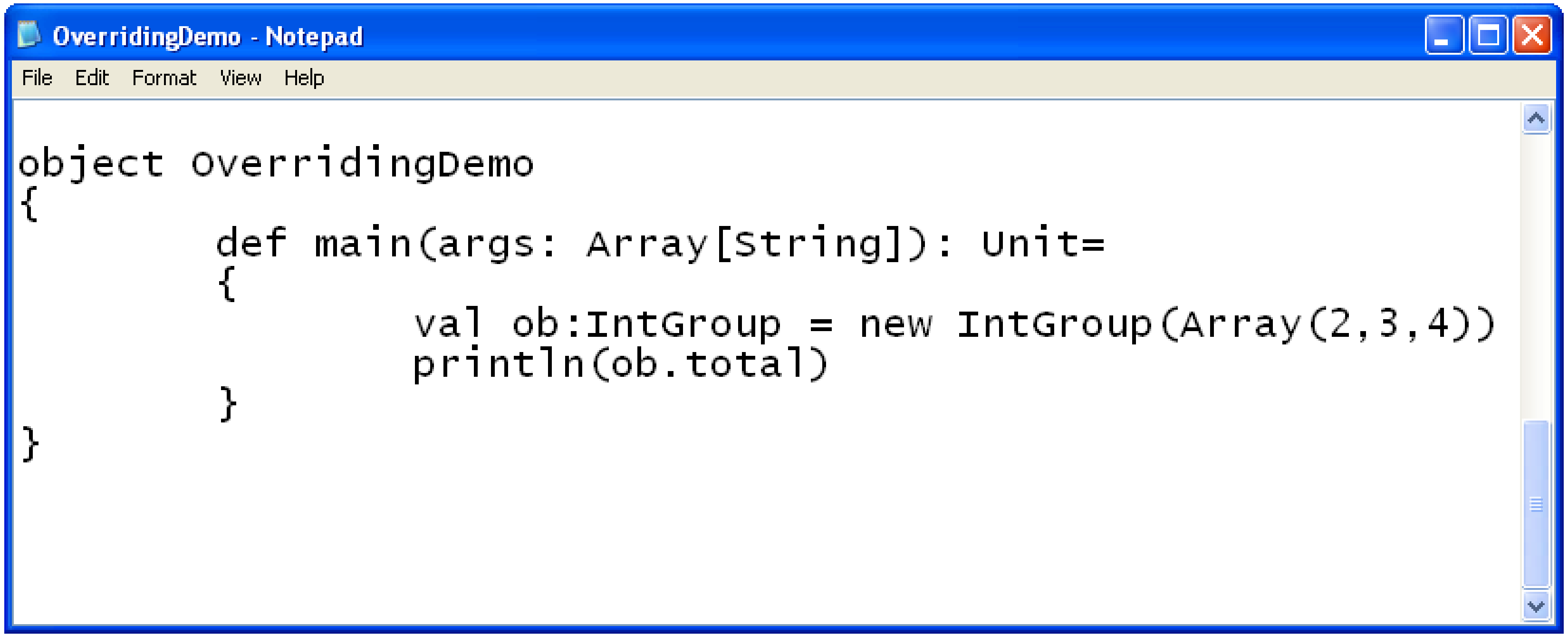
- We access fields and methods using the same syntax. Fields and methods belong to the same namespace.
- A field can override a parameter-less method. This way we can change the implementation from a method to a field.

Overriding Methods & Fields



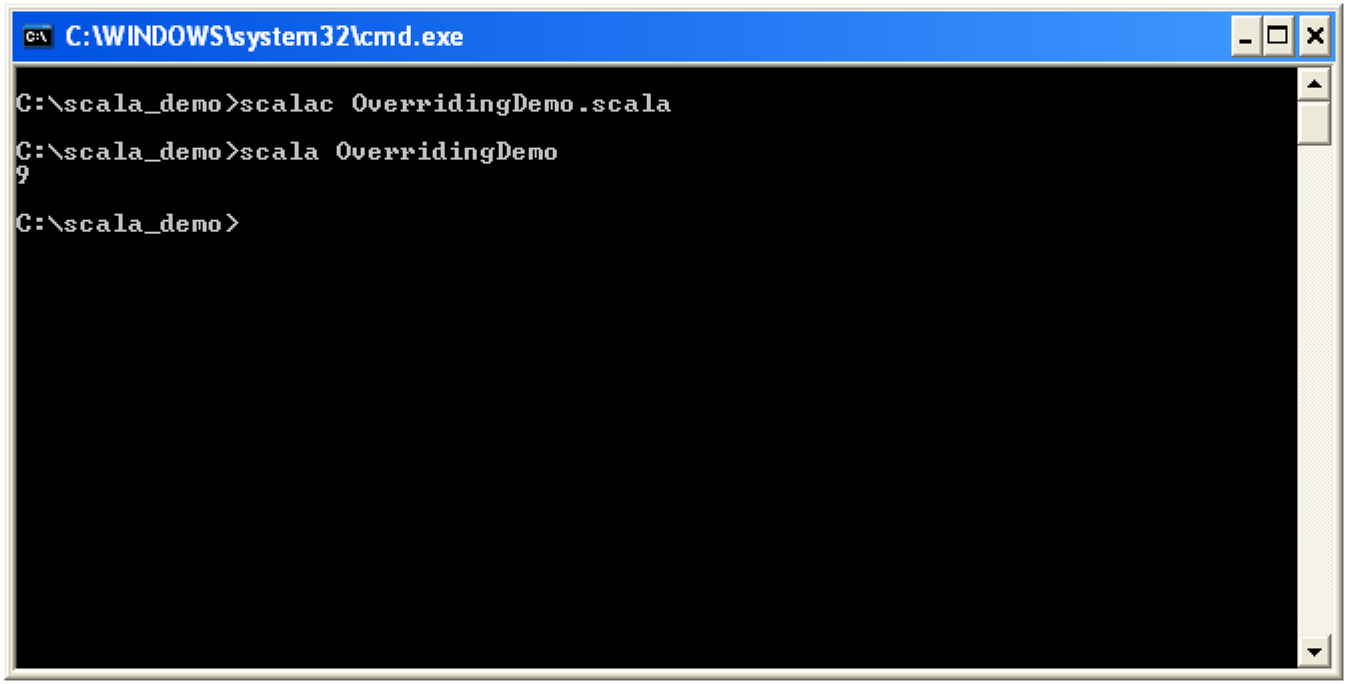
```
abstract class Item
{
        def content: Array[Int]
}

class IntGroup(cont: Array[Int]) extends Item
{
        val content: Array[Int] = cont
        def total:Int =
        {
                var i:Int = 0
                var sum:Int = 0
                while(i<content.length)
                {
                        sum = sum + content(i)
                        i=i+1
                }
                sum
        }
}
```

Overriding Methods & Fields



```
object OverridingDemo
{
        def main(args: Array[String]): Unit=
        {
                val ob:IntGroup = new IntGroup(Array(2,3,4))
                println(ob.total)
        }
}
```

Overriding Methods & Fields



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

C:\scala_demo>scala OverridingDemo
9

C:\scala_demo>
```

Overriding Methods & Fields

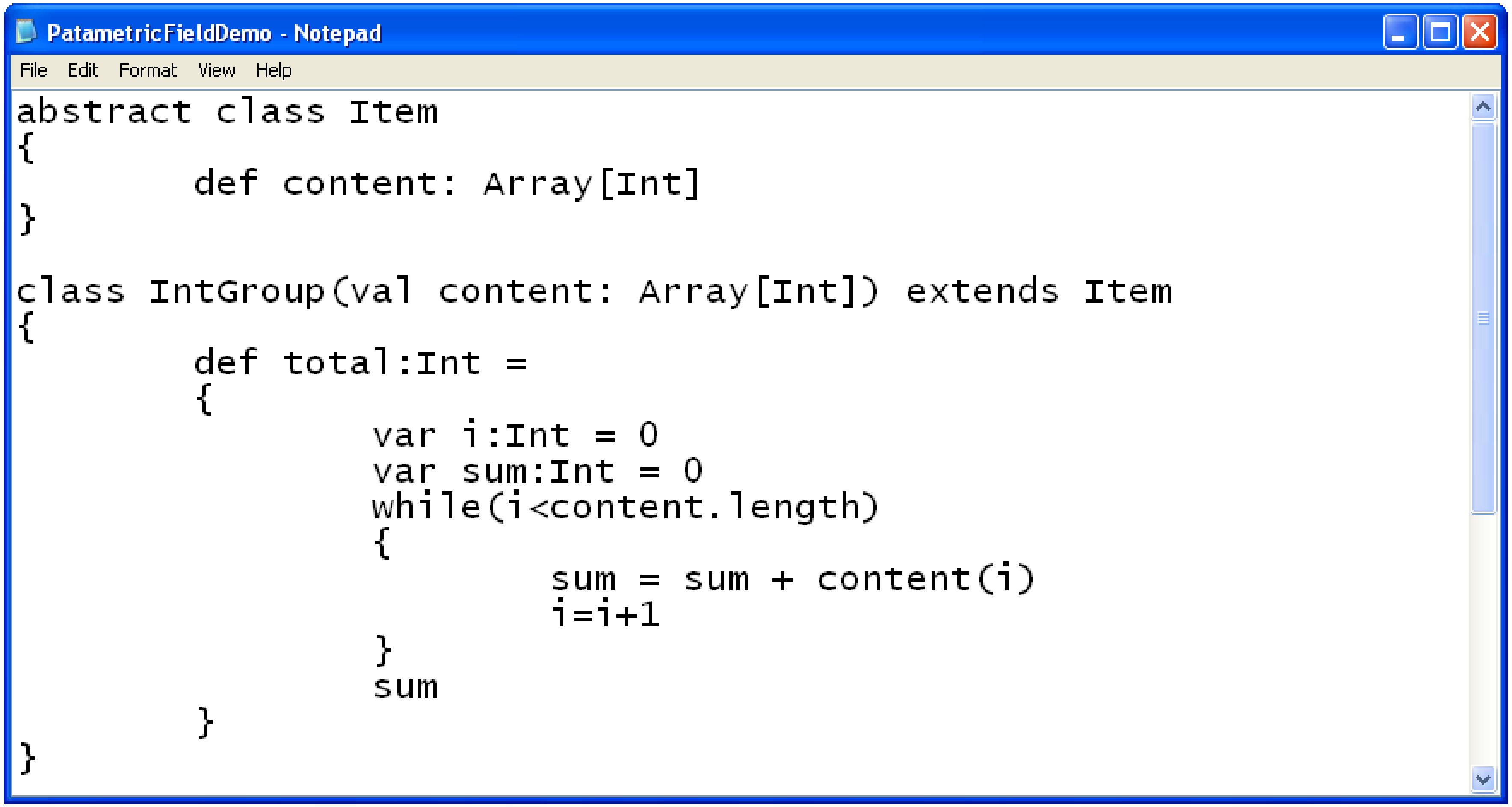
- Scala doesn't allow us to define within the same class a field and a method with the same name.
- Java has four name spaces: fields, methods, types and packages. Scala has two. Values (fields,methods,packages and singleton) and types (class and traits).

Parametric Fields

- We can define a class parameter with the field it targets in one single definition.

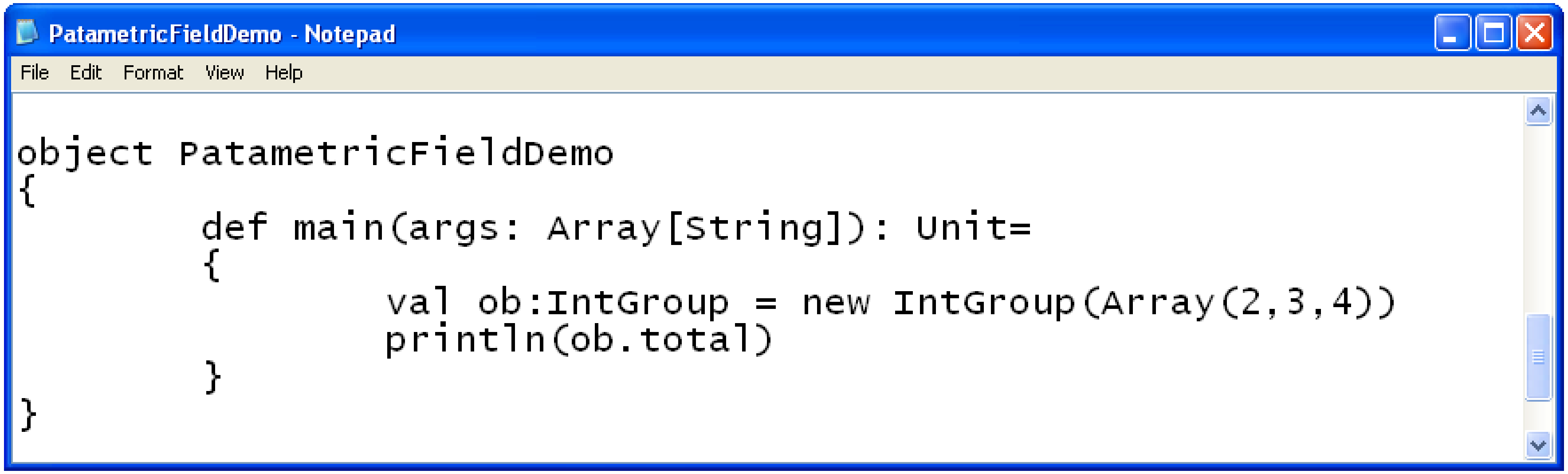
```
...
class IntGroup(val content: Array[Int]) extends Item
{
    def total:Int =
    {
          ...
    }
}
...                                                   w
```

Parametric Fields



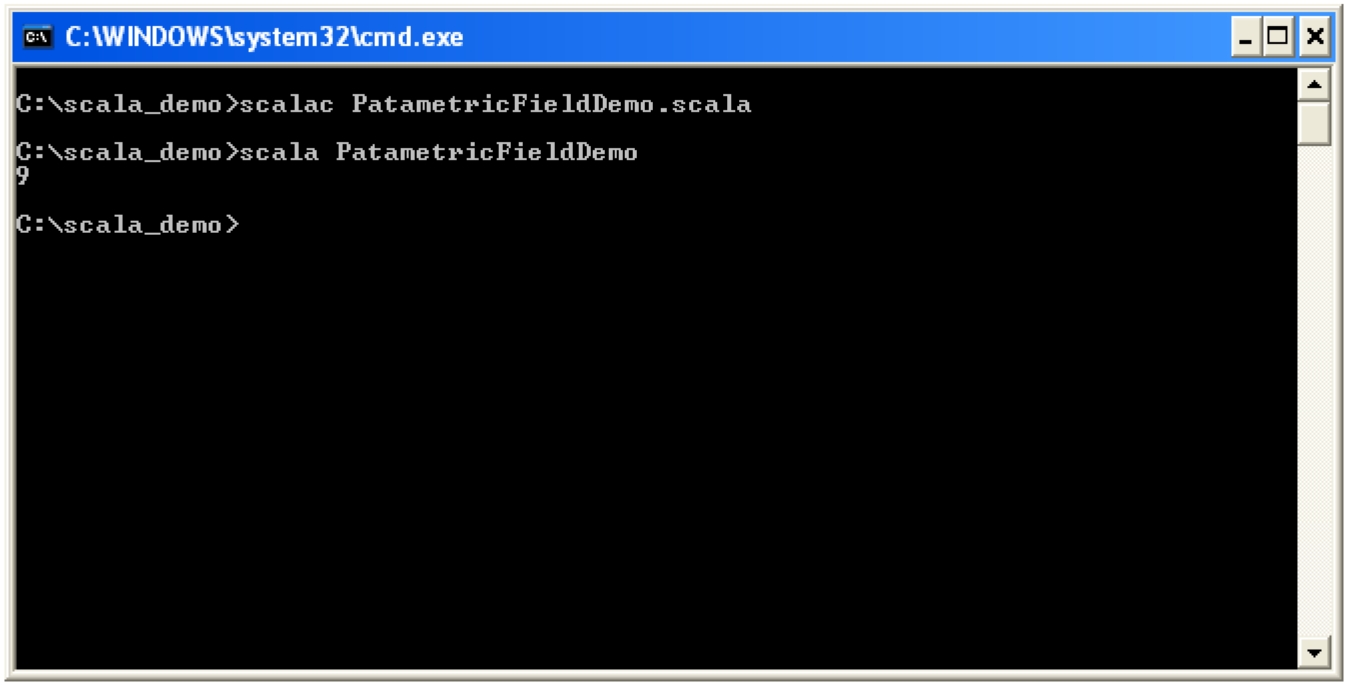
```
abstract class Item
{
        def content: Array[Int]
}

class IntGroup(val content: Array[Int]) extends Item
{
        def total:Int =
        {
                var i:Int = 0
                var sum:Int = 0
                while(i<content.length)
                {
                        sum = sum + content(i)
                        i=i+1
                }
                sum
        }
}
```

Parametric Fields



```
object PatametricFieldDemo
{
        def main(args: Array[String]): Unit=
        {
                val ob:IntGroup = new IntGroup(Array(2,3,4))
                println(ob.total)
        }
}
```

Parametric Fields



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

C:\scala_demo>scala PatametricFieldDemo
9

C:\scala_demo>
```

Invoking Super Class Constructors

❖ We can place a call to specific super class constructor by placing the argument \ arguments we want to pass in parentheses following the name of the superclass.

...

```
class IntGroup(val content: Array[Int]) extends Item(“mygrp”)
{
    ...
}
...
```

The `override` Modifier

- When overriding a concrete member in a parent class we must use the override modifier. When overriding an abstract member this modifier is optional.

Polymorphism

- The Scala programming language supports polymorphism.

Final Class

- Adding the final modifier to specific class will ensure that it won't be possible to extend it.

```
final class Box
{
    ...
}
```

Final Method

- Adding the final modifier to specific method will ensure that it won't be possible to override it.

```
class Box
{
    ...
    final def getId:Int = id
    ...
}
```