# Example 51

# Classes

# Classes

We will not spend any time at all explaining
what classes are - just how Scala classes work

## Let's start with the common, simple operations we need to perform

# Classes

## Let's start with the common, simple operations we need to perform

Define a class

Instantiate a class

Use a class value/variable

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

These values are known as class parameters - think of them as parameters to the default constructor

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

The class parameters can be used to initialise fields (values and variables inside a class)

# Define a class

```
class Rectangle(val length:Double, val breadth:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=$breadth"
}
```

To make a class parameter a field, simply add the words val or var before it

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

Methods work as you would expect - remember they can be invoked either with dot infix notation or suffix notation

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=$breadth"
}
```

check out the override keyword, since toString is defined in Object

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

check out the override keyword, since toString is defined in Object

# Define a class

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

fields in the class can be used in methods, say you would expect..

# Define a class

fields in the class can be used in methods, say you would expect..

btw, everything is **public** by default (more on access modifiers later)

# Classes

## Let's start with the common, simple operations we need to perform

- Define a class
- Instantiate a class
- Use a class value/variable

# Instantiate a class

```
val rect = new Rectangle(4,5)
```

Nothing very new here

# Instantiate a class

```
val rectList = List(new Rectangle(3,4),rect)
```

## Using instances of a class in a container

# Classes

## Let's start with the common, simple operations we need to perform

- Define a class
- Instantiate a class
- Use a class value/variable

# Use a class value/variable

```
scala> rectList.map( _.getArea)
res0: List[Double] = List(12.0, 20.0)

scala> rectList.foreach( (x:Rectangle) => {println(x.toString)})

I am a rectangle, l=3.0,b=4.0
I am a rectangle, l=4.0,b=5.0
```

Once instantiated, using class variables is easy

# Use a class value/variable

```
scala> rectList.map( _.getArea)
res0: List[Double] = List(12.0, 20.0)

scala> rectList.foreach( (x:Rectangle) => {println(x.toString)})

I am a rectangle, l=3.0,b=4.0
I am a rectangle, l=4.0,b=5.0
```

Once instantiated, using class variables is easy

# Use a class value/variable

```
scala> rectList.map( _.getArea)
res0: List[Double] = List(12.0, 20.0)

scala> rectList.foreach( (x:Rectangle) => {println(x.toString)})

I am a rectangle, l=3.0,b=4.0
I am a rectangle, l=4.0,b=5.0
```

Once instantiated, using class variables is easy

# Use a class value/variable

```
scala> rectList.map( _.getArea)
res0: List[Double] = List(12.0, 20.0)

scala> rectList.foreach( (x:Rectangle) => {println(x.toString)})

I am a rectangle, l=3.0,b=4.0
I am a rectangle, l=4.0,b=5.0
```

Once instantiated, using class variables is easy

# Use a class value/variable

```
scala> rectList.map( _.getArea)
res0: List[Double] = List(12.0, 20.0)

scala> rectList.foreach( (x:Rectangle) => {println(x.toString)})
```

```
I am a rectangle, l=3.0,b=4.0
I am a rectangle, l=4.0,b=5.0
```

Once instantiated, using class variables is easy

# Classes

## Let's start with the common, simple operations we need to perform

- Define a class
- Instantiate a class
- Use a class value/variable

# Example 52

# Primary v Auxiliary Constructors

# Primary v Auxiliary Constructors

The primary constructor is simply the code that deals with class parameters

Auxiliary constructors are additional methods named `this`

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

The primary constructor is simply the code that deals with class parameters

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

The primary constructor is simply the code that deals with class parameters

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

The primary constructor is simply the code that deals with class parameters

# Primary v Auxiliary Constructors

The primary constructor is simply the code that deals with class parameters

Auxiliary constructors are additional methods named `this`

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

Auxiliary constructors are additional methods named `this`

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

Auxiliary constructors usually invoke the primary constructor, especially when inheritance is involved

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

Auxiliary constructors usually invoke the primary constructor, especially when inheritance is involved

# Primary v Auxiliary Constructors

```
class Rectangle(l:Double,b:Double) {
 val length:Double = l
 val breadth:Double = b
 def this(s:Double) = this(s,s)
 def getArea:Double = l * b
 override def toString = s"I am a rectangle, l=$length, b=
$breadth"
}
```

# Primary v Auxiliary Constructors

Btw, the primary constructor can take default values too

# Primary v Auxiliary Constructors

## Btw, the primary constructor can take default values too

```
class Fraction(n:Double,d:Double=1) {
  val numer:Double = n
  val denom:Double = d
  def getFraction = numer/denom

}
```

# Primary v Auxiliary Constructors

## Btw, the primary constructor can take default values too

```
class Fraction(n:Double,d:Double=1) {
  val numer:Double = n
  val denom:Double = d
  def getFraction = numer/denom

}
```

# Primary v Auxiliary Constructors

## Btw, the primary constructor can take default values too

```
class Fraction(n:Double,d:Double=1) {
  val numer:Double = n
  val denom:Double = d
  def getFraction = numer/denom

}
```

## This is equivalent to an auxiliary constructor, in a sense

# Primary v Auxiliary Constructors

## Btw, the primary constructor can take default values too

```
class Fraction(n:Double,d:Double=1) {
  val numer:Double = n
  val denom:Double = d
  def getFraction = numer/denom

}
```

# Example 53

# Inheritance from Classes

# Inheritance

## From classes

Like Java inheritance from classes - nothing too major

## From traits

Like Java inheritance from interfaces - but with a number of little twists

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

## A base class Shape

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
}
```

## A derived class Rectangle

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
}
```

## A derived class Rectangle

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
}
```

The base class constructor is invoked by the primary constructor of the derived class

# Inheritance from Classes

The base class constructor is invoked by the primary constructor of the derived class

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
}
```

## Remember, by default, everything is public in Scala

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") exte
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length, b=$breadth"
}
```

```
class Square(s:Double) extends Rectangle(s,s,"Square") {

}
```

## And yet another derived class Square

# Inheritance from Classes

```
class Shape(name:String) {
  val shapeName = name
  override def toString= s"I am a $shapeName"
}
```

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") exte
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length, b=$breadth"
}
```

```
class Square(s:Double) extends Rectangle(s,s,"Square") {

}
```

## And yet another derived class Square

# Inheritance from Classes

```
scala> val s:Shape = new Rectangle(10,5)
s: Shape = I am a Rectangle, l=10.0, b=5.0

scala> val r:Rectangle = new Square(5)
r: Rectangle = I am a Square, l=5.0, b=5.0
```

Base classes values can hold derived class "objects"

# Inheritance from Classes

```
scala> val s:Shape = new Rectangle(10,5)
s: Shape = I am a Rectangle, l=10.0, b=5.0

scala> val r:Rectangle = new Square(5)
r: Rectangle = I am a Square, l=5.0, b=5.0
```

Base classes values can hold derived class "objects"

# Inheritance from Classes

Base classes values can hold derived class "objects"

But the reverse is not true

All just like in Java

# Example 54

# Abstract Classes

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

Pretty obviously, the Shape class has no business being instantiated

# Abstract Classes

Pretty obviously, the Shape class has no business being instantiated

```
scala> val s:Shape = new Shape("Circle")
<console>:12: error: class Shape is abstract; cannot be instantiated
       val s:Shape = new Shape("Circle")
                     ^
```

So, mark it abstract, and add an unimplemented method

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

So, mark it abstract, and add an unimplemented method

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

So, mark it abstract, and add an unimplemented method

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

So, mark it abstract, and add an unimplemented method

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
  def getArea = l*b
}
```

The derived class does not change

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
  def getArea = l*b
}
```

The derived class does not change

# Abstract Classes

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

```
class Rectangle(l:Double, b:Double,
shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length,
b=$breadth"
  def getArea = l*b
}
```

The unimplemented method is defined, of course!

# Abstract Classes

```
scala> val shape:Shape = new Rectangle(5,5)
shape: Shape = I am a Rectangle, l=5.0, b=5.0

scala> val shape:Shape = new Square(5)
shape: Shape = I am a Square, l=5.0, b=5.0

scala> val rect:Rectangle = new Square(5)
rect: Rectangle = I am a Square, l=5.0, b=5.0
```

Abstract classes values can hold derived class "objects" of course

# Example 55

# Anonymous Classes

# Anonymous Classes

As in Java, anonymous classes are a quick way to instantiate classes with abstract methods

Define the abstract methods between a pair of curly braces, and instantiate using new!

# Anonymous Classes

As in Java, anonymous classes are a quick way to instantiate classes with abstract methods

```
abstract class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
  def getArea:Double
}
```

# Anonymous Classes

## Define the abstract methods between a pair of curly braces, and instantiate using new!

```
val someShape:Shape = new Shape("Irregular shape") {
  def getArea = 100
}
```

# Anonymous Classes

Define the abstract methods between a pair of curly braces, and instantiate using new!

```
val someShape:Shape = new Shape("Irregular shape") {
  def getArea = 100
}
```

this can even be done using values!

# Anonymous Classes

Define the abstract methods between a pair of curly braces, and instantiate using new!

```
val someShape:Shape = new Shape("Irregular shape") {
  def getArea = 100
}
```

this can even be done using values!

# Anonymous Classes

Define the abstract methods between a pair of curly braces, and instantiate using new!

```
val someShape:Shape = new Shape("Irregular shape") {
  def getArea = 100
}
```

This is a perfectly legit instantiation of that abstract class now!

# Anonymous Classes

Define the abstract methods between a pair of curly braces, and instantiate using new!

```
scala> val someShape:Shape = new Shape("Irregular shape") {
     |     def getArea = 100
     | }
someShape: Shape = I am a Irregular shape

scala> someShape.getArea
res13: Double = 100.0
```

This is a perfectly legit instantiation of that abstract class now!

# Example 56

# Type Parameters

# Type Parameters

As in Java and C++, type parameters help make a class generic

This is very similar to how it works in Java, so we won't dwell on it a whole lot

# Type Parameters

## help make a class generic

```
class AreaCalculator[T](s:T) {
  val shape:T = s
  override def toString = shape.toString
}
```

## [ ] in Scala versus <> in Java - thats the biggest difference :-)

# Type Parameters

## help make a class generic

```
class AreaCalculator[T](s:T) {
  val shape:T = s
  override def toString = shape.toString
}
```

[ ] in Scala versus <> in Java - thats the biggest difference :-)

# Type Parameters

## help make a class generic

```
class AreaCalculator[T](s:T) {
  val shape:T = s
  override def toString = shape.toString
}
```

## Use the type parameter in your class parameters, fields, method definitions..

# Type Parameters

## help make a class generic

Use the type parameter in your class parameters, fields, method definitions..

```
val a = new AreaCalculator[Shape](shape)
```

## Instantiate the value using a specific type, and []

# Type Parameters

## help make a class generic

```
val a = new AreaCalculator[Shape](shape)
```

## Instantiate the value using a specific type, and []

# Example 57

# Lazy Values

# Lazy Values

Exactly as their name would suggest, these are member fields that are lazily instantiated

this is great for very heavy-duty objects that might never get used at all

Fields not marked lazy are eagerly instantiated..

# Lazy Values

## member fields that are lazily instantiated

```
class LazyRectangle(l:Double, b:Double) {
  val length = l
  val breadth = b
  val getArea = {println("initialising area"); l*b}
  lazy val perimeter = {println("initialising perimeter");
2*(length+breadth)}
}
```

## Fields not marked lazy are eagerly instantiated..

# Lazy Values

## member fields that are lazily instantiated

```
class LazyRectangle(l:Double, b:Double) {
  val length = l
  val breadth = b
  val getArea = {println("initialising area"); l*b}
  lazy val perimeter = {println("initialising perimeter");
2*(length+breadth)}
}
```

## Fields not marked lazy are eagerly instantiated..

# Lazy Values

## member fields that are lazily instantiated

```
scala> val r = new LazyRectangle(5,20)
initialising area
r: LazyRectangle = LazyRectangle@2d7d77d3
```

## Fields not marked lazy are eagerly instantiated..

# Lazy Values

## member fields that are lazily instantiated

```
scala> val r = new LazyRectangle(5,20)
initialising area
r: LazyRectangle = LazyRectangle@2d7d77d3
```

## Fields not marked lazy are eagerly instantiated..

# Lazy Values

Exactly as their name would suggest, these are member fields that are lazily instantiated

this is great for very heavy-duty objects that might never get used at all

Fields not marked lazy are eagerly instantiated..

# Lazy Values

## member fields that are lazily instantiated

```
class LazyRectangle(l:Double, b:Double) {
  val length = l
  val breadth = b
  val getArea = {println("initialising area"); l*b}
  lazy val perimeter = {println("initialising perimeter");
2*(length+breadth)}
}
```

## Simply mark the field as lazy

# Lazy Values

## member fields that are lazily instantiated

```
class LazyRectangle(l:Double, b:Double) {
  val length = l
  val breadth = b
  val getArea = {println("initialising area"); l*b}
  lazy val perimeter = {println("initialising perimeter");
2*(length+breadth)}
}
```

## Simply mark the field as lazy

# Lazy Values

## member fields that are lazily instantiated

```
class LazyRectangle(l:Double, b:Double) {
  val length = l
  val breadth = b
  val getArea = {println("initialising area"); l*b}
  lazy val perimeter = {println("initialising perimeter");
2*(length+breadth)}
}
```

## Simply mark the field as lazy

# Lazy Values

## member fields that are lazily instantiated

```
scala> val r = new LazyRectangle(5,20)
initialising area
r: LazyRectangle = LazyRectangle@2d7d77d3
```

## and the field is not initialised during instantiation

# Lazy Values

member fields that are lazily instantiated

and the field is not initialised during instantiation

```
scala> r.perimeter
initialising perimeter
res15: Double = 50.0
```

only when the field is actually referenced

# Lazy Values

member fields that are lazily instantiated

and the field is not initialised during instantiation

```
scala> r.perimeter
initialising perimeter
res15: Double = 50.0
```

only when the field is actually referenced

# Lazy Values

member fields that are lazily instantiated

and the field is not initialised during instantiation

```
scala> r.perimeter
initialising perimeter
res15: Double = 50.0
```

only when the field is actually referenced

# Example 58

# Default Methods with apply

# Default Methods with apply

This is a rather peculiar Scala construct -

methods named apply can be invoked without the method name

these methods are known as default methods, since merely the value name is enough to invoke them!

# Default Methods with apply

```
class Rectangle(l:Double, b:Double) {
  val length = l
  val breadth = b
}
```

# A simple class definition

# Default Methods with apply

```
class AreaCalculator {
  def apply(s:Rectangle) = s.length * s.breadth
}
```

And another simple class with a method named apply

# Default Methods with apply

```
class AreaCalculator {
  def apply(s:Rectangle) = s.length * s.breadth
}
```

And another simple class with a method named apply

# Default Methods with apply

```
class AreaCalculator {
  def apply(s:Rectangle) = s.length * s.breadth
}
```

## And another simple class with a method named apply

# Default Methods with apply

A simple class with a method named **apply**

we can instantiate this class, and invoke the apply method without naming it

# Default Methods with apply

## we can instantiate this class, and invoke the apply method without naming it

```
val lazyRect = new LazyRectangle(4,5)
val areaCalc = new AreaCalculator()
```

```
scala> areaCalc(lazyRect)
res17: Double = 20.0
```

# Default Methods with apply

## we can instantiate this class

```
val lazyRect = new LazyRectangle(4,5)
val areaCalc = new AreaCalculator()
```

# Default Methods with apply

## invoke the apply method without naming it

```
scala> areaCalc(lazyRect)
res17: Double = 20.0
```

# The term
# Default Method

is used for such methods since they are triggered
by "applying" the value as if it were a function

# Example 59

# Operators

# Operators

In Scala, as in C++, but unlike in Java

operators can be overloaded

operators are merely methods, with names of symbols such as +,- and so on

operators are merely methods, with names of symbols such as +,- and so on

in many contexts, these are more intuitive to use than function names

the other reason operators are easy to use is: Scala allows methods to be invoked this way

# Given a list

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
```

## `.contains` tests for a specific element

```
weekDays.contains(“Mon”)  // infix dot notation

weekDays contains “Mon”   // operator notation

Boolean = true
```

```
// infix dot notation

        weekDays.contains(“Mon”)

// operator notation

        weekDays contains “Mon”

// Same result

                Boolean = true
```

# Operators

## Consider a complex number class

```
class ComplexNumber(r:Double, i:Double) {
  val realPart = r
  val imaginaryPart = i

  def +(that: ComplexNumber): ComplexNumber =
      new ComplexNumber(
        this.realPart + that.realPart,
        this.imaginaryPart + that.imaginaryPart
        )

  override def toString  = s"($realPart,$imaginaryPart)"
}
```

## it would be great to be able to add complex numbers using +

# add complex numbers using +

```
class ComplexNumber(r:Double, i:Double) {
  val realPart = r
  val imaginaryPart = i

  def +(that: ComplexNumber): ComplexNumber =
      new ComplexNumber(
        this.realPart + that.realPart,
        this.imaginaryPart + that.imaginaryPart
        )

  override def toString  = s"($realPart,$imaginaryPart)"
}
```

# add complex numbers using +

```
def +(that: ComplexNumber): ComplexNumber =
    new ComplexNumber(
      this.realPart + that.realPart,
      this.imaginaryPart + that.imaginaryPart
      )
```

Other than the method name, nothing about this method is unusual

# add complex numbers using +

```
def +(that: ComplexNumber): ComplexNumber =
    new ComplexNumber(
      this.realPart + that.realPart,
      this.imaginaryPart + that.imaginaryPart
      )
```

Other than the method name,
nothing about this method is unusual

# Operators

```
scala> val c1 = new ComplexNumber(5,10)
c1: ComplexNumber = (5.0,10.0)

scala> val c2 = new ComplexNumber(1,3)
c2: ComplexNumber = (1.0,3.0)

scala> val c3 = c1 + c2
c3: ComplexNumber = (6.0,13.0)
```

check out how intuitive this syntax is

# Operators

## Instantiate 2 values the usual way

```
scala> val c1 = new ComplexNumber(5,10)
c1: ComplexNumber = (5.0,10.0)

scala> val c2 = new ComplexNumber(1,3)
c2: ComplexNumber = (1.0,3.0)
```

## check out how intuitive this syntax is

# Operators

then add them using the operator+

```
scala> val c3 = c1 + c2
c3: ComplexNumber = (6.0,13.0)
```

check out how intuitive this syntax is

# Example 60

# Access Modifiers

# Access Modifiers

In Scala, by default
all fields are public

but can be marked private and
protected to restrict access

As in Java, members and classes can be marked
final, and classes can be marked sealed

# Access Modifiers

## In Scala, by default all fields are public

```
class Shape(name:String) {
  protected val shapeName = name
  override def toString = s"I am a $shapeName"
}
```

## but can be marked private and protected to restrict access

# Access Modifiers

## public and protected fields are accessible in all derived classes

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") extends Shape(shapeName)
  val length = l
  val breadth = b
  private val diagLen = math.sqrt(length*length + breadth*breadth)
  override def toString = s"I am a $shapeName,diagonal = $diagLen"
}
```

## (the same semantics as in Java)

# Access Modifiers

## public and protected fields are accessible in all derived classes

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") extends
Shape(shapeName) {
  val length = l
  val breadth = b
  private val diagLen = math.sqrt(length*length + breadth*breadth)
  override def toString = s"I am a $shapeName,diagonal = $diagLen"
}
```

## (the same semantics as in Java)

# Access Modifiers

## public and protected fields are accessible in all derived classes

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") extends Shape(shapeName)
  val length = l
  val breadth = b
  private val diagLen = math.sqrt(length*length + breadth*breadth)
  override def toString = s"I am a $shapeName,diagonal = $diagLen"
}
```

```
class Shape(name:String) {
  protected val shapeName = name
  override def toString = s"I am a $shapeName"
}
```

# Access Modifiers

## public and protected fields are accessible in all derived classes

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") extends Shape(shapeName)
  val length = l
  val breadth = b
  private val diagLen = math.sqrt(length*length + breadth*breadth)
  override def toString = s"I am a $shapeName,diagonal = $diagLen"
}
```

## marking a field private ensures derived types can not access it..

# Access Modifiers

```
class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  private val diagLen = math.sqrt(length*length + breadth*breadth)
  override def toString = s"I am a $shapeName,diagonal = $diagLen"
}
```

marking a field private ensures derived types can not access it..

```
class Square(s:Double) extends Rectangle(s,s,"Square") {
   // override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal =
$diagLen"
 override def toString = s"I am a $shapeName, l=$length, b=$breadth"
}
```

# Access Modifiers

## marking a field private ensures derived types can not access it..

```
class Square(s:Double) extends Rectangle(s,s,"Square") {
   // override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal =
$diagLen"
 override def toString = s"I am a $shapeName, l=$length, b=$breadth"
}
```

# Access Modifiers

## marking a field private ensures derived types can not access it..

```
class Square(s:Double) extends Rectangle(s,s,"Square") {
      override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal =
$diagLen"
 override def toString = s"I am a $shapeName, l=$length, b=$breadth"
}
```

```
scala> class Square(s:Double) extends Rectangle(s,s,"Square") {
     |     override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal = $diagLen"
     | }
<console>:15: error: not found: value diagLen
           override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal = $diagLen"
                                                                                          ^
```

# Access Modifiers

## marking a field private ensures derived types can not access it..

```
scala> class Square(s:Double) extends Rectangle(s,s,"Square") {
     |     override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal = $diagLen"
     | }
<console>:15: error: not found: value diagLen
            override def toString = s"I am a $shapeName, l=$length, b=$breadth, diagonal = $diagLen"
                                                                                            ^
```

# Access Modifiers

In Scala, by default
all fields are public

but can be marked private and
protected to restrict access

As in Java, members and classes can be marked
final, and classes can be marked sealed

members and classes can be marked final

final members or classes can not be overridden at all in subclasses

classes can be marked sealed

sealed classes must be located in the same file as the parent class

# Access Modifiers

In Scala, by default
all fields are public

but can be marked private and
protected to restrict access

As in Java, members and classes can be marked
final, and classes can be marked sealed

# Example 61

# Singleton Objects

# Singleton Objects

In Scala, Static member variables and methods cannot be defined inside a class

Scala provides a special construct for defining static variables/behaviour

# Singleton Objects

Singleton objects are a way to

1. Define a class that's only going to be used to create 1 object

2. Define static behavior and variables

# Singleton Objects

```
object Rational{

  val inf = "infinity"

  def divideByZero(x:Int): Unit ={
    println("Not defined")
  }

}
```

Use the object keyword to create a Singleton object

# Singleton Objects

```
object Rational{
  val inf = "infinity"

  def divideByZero(x:Int): Unit ={
    println("Not defined")
  }

}
```

Use these methods and variables as if they are static members of Rational

```
println(Rational.inf);

Rational.divideByZero(5);
```

# Singleton Objects

The main method that's used to run a program must be static i.e. enclosed in a Singleton object

```
object HelloWorld {

   def main (args: Array[String]){

     println("Hello, this is SCALA!")

   }

}
```

# Singleton Objects

An object can extend a class, but a class can not extend an object

# Singleton Objects

Objects do not take "class parameters" btw

# Example 62

# Companion Objects

# Companion Objects

A singleton object and a class with the same name have a special relationship in Scala

the object is said to be a companion object of the class

# Companion Objects

A singleton object and a class with the same name have a special relationship in Scala

```
object NYTimesAccount {
}

class NYTimesAccount {
}
```

the object is said to be a companion object of the class

# Companion Objects

```
object NYTimesAccount {
}

class NYTimesAccount {
}
```

the class and the companion object need to be defined in the same file

# Companion Objects

the class and the companion object need to be defined in the same file

```
object NYTimesAccount {
}

class NYTimesAccount {
}
```

and they can access each others private and protected fields

# Companion Objects

in the absence of static methods, companion objects are the standard way to implement

## the factory pattern

in Scala

# the factory pattern in Scala

```
object NYTimesAccount {

  private val userName = "vitthal"
  private val password = "boo"

  def apply() = new NYTimesAccount

}

class NYTimesAccount {
}
```

# the factory pattern

## in Scala

```
object NYTimesAccount {

  private val userName = "vitthal"
  private val password = "boo"

  def apply() = new NYTimesAccount

}

class NYTimesAccount {
}
```

# the factory pattern

## in Scala

```
object NYTimesAccount {

  private val userName = "vitthal"
  private val password = "boo"

  def apply() = new NYTimesAccount
}

class NYTimesAccount {
}
```

this apply function is where the magic happens

# the factory pattern

## in Scala

this apply function is where the magic happens

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
```

btw, this is exactly how Lists are instantiated!

# the factory pattern in Scala

btw, this is exactly how Lists are instantiated!

```
val weekDays = List("Mon","Tue","Wed","Thu","Fri")
```

The `List` class has a companion `List` object with an apply method!

# the factory pattern in Scala

```
object NYTimesAccount {
}

class NYTimesAccount {
  private val accountDetails = Map(
  "user" -> NYTimesAccount.userName,
  "password" -> NYTimesAccount.password
  )
  println(s"Hi " + accountDetails("user"))
}
```

# the factory pattern in Scala

```
object NYTimesAccount {
}

class NYTimesAccount {
  private val accountDetails = Map(
  "user" -> NYTimesAccount.userName,
  "password" -> NYTimesAccount.password
  )
  println(s"Hi " + accountDetails("user"))
}
```

# the factory pattern in Scala

```
object NYTimesAccount {
}

class NYTimesAccount {
  private val accountDetails = Map(
  "user" -> NYTimesAccount.userName,
  "password" -> NYTimesAccount.password
  )
  println(s"Hi " + accountDetails("user"))
}
```

# Companion Objects

in the absence of static methods, companion objects are the standard way to implement

## the factory pattern

in Scala

# Example 63

# Traits

# Traits

Scala does not have Interfaces

Traits are the closest thing
to Interfaces in Scala

# Traits

Traits can have implemented methods

(like interface default methods in Java 8)

# Traits

Like interfaces, Traits can't be instantiated directly

Classes can inherit from
only 1 Class/Abstract Class

Classes can inherit from
any number of traits

This is the main difference
between Classes and Traits

```
abstract class Animal {
  def speak
}
```

# An abstract class

```
abstract class Animal {
  def speak
}
```

```
trait FourLeggedAnimal {
  def walk
  def run
}
```

# A trait with abstract methods

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}


trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}
```

A trait with implemented methods

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}

trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}

class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

A class that inherits from 1 abstract class and 2 traits

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}
```

```scala
trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}
```

```scala
class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

Use the extends keyword with the first inheritance (either trait/class)

```scala
abstract class Animal {
  def speak
}
```

```scala
trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}
```

```scala
trait FourLeggedAnimal {
  def walk
  def run
}
```

```scala
class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

Use the with keyword for any inheritances that follow

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}

trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}

class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

If a class inherits from both another class and traits, use extends with the class and with for all the subsequent traits

# Example 64

# Case Classes

# Case Classes

Case classes are a syntactic shortcut provided by Scala

They represent a way to define a class with a single line of code

```
case class Person(firstName:String, lastName:String)
```

# Case Classes

this one line generates a class with automatically generated methods

```
case class Person(firstName:String, lastName:String)
```

and a companion object with an apply method

# Case Classes

```
val person1 = Person("Vitthal","Srinivasan")
val person2 = Person("Janani","Ravi")
```

Now, we can instantiate this class using that companion object + apply method

# Case Classes

## Now, we can instantiate this class using that companion object + apply method

```
val person1 = Person("Vitthal","Srinivasan")
```

```
person1 match {
  case Person("Vitthal","Srinivasan") => "Humpty-Dumpty"
  case x => x.firstName
}
```

## and also use it in interesting match statements like this one

# Case Classes
## interesting match statements

```
val person1 = Person("Vitthal","Srinivasan")

person1 match {
  case Person("Vitthal","Srinivasan") => "Humpty-Dumpty"
  case x => x.firstName
}
```

# Case Classes
## interesting match statements

```
val person1 = Person("Vitthal","Srinivasan")

person1 match {
  case Person("Vitthal","Srinivasan") => "Humpty-Dumpty"
  case x => x.firstName
}
```

# Case Classes
## interesting match statements

```
val person1 = Person("Vitthal","Srinivasan")

person1 match {
  case Person("Vitthal","Srinivasan") => "Humpty-Dumpty"
  case x => x.firstName
}
```

# Case Classes
## interesting match statements

```
val person1 = Person("Vitthal","Srinivasan")

person1 match {
  case Person("Vitthal","Srinivasan") => "Humpty-Dumpty"
  case x => x.firstName
}
```

# Case Classes

Case classes are a syntactic shortcut provided by Scala

They represent a way to define a class with a single line of code

```
case class Person(firstName:String, lastName:String)
```

# Example 65

# Self Types

# Self Types

this is a new bit of functionality available in Scala:

the java equivalent is makes this easier to understand

# Self Types

this is a new bit of functionality available in Scala:

the java equivalent

If a class implements an interface X, it must also implement another interface Y

# Self Types

## the java equivalent

If a class implements an interface X, it must also implement another interface Y

or

If a class implements an interface X, it must also extend an abstract class Z

# Java simply does not offer an easy way to specify these semantics

(without actually modifying the interfaces and classes Y,Z)

# But Scala does

# Self Types

the java equivalent

If a class inherits a self type, it must be subtype of the self type

or

If a class inherits a self type, it must also be a subclass of

# Self Types

If a class extends a trait X, it must also be **mixed in with another trait Y**

"mixing traits" is the term for getting a type to satisfy "is-a X" and "is-a Y"

this is done in the trait X via a self type

# Self Types

Given a class **Person**

```
case class Person(firstName:String, lastName:String)
```

And a trait **DormResident**

```
trait DormResident {
  val livesOnCampus = true
}
```

We want any class that extends **DormResident** to also be a **Person**

# Self Types

We want any class that extends **`DormResident`** to also be a **`Person`**

```
trait DormResident { self: Person =>
  val livesOnCampus = true
}
```

add this bit of syntax to the trait

# Self Types

We want any class that extends **`DormResident`** to also be a **`Person`**

```
trait DormResident { self: Person =>
  val livesOnCampus = true
}
```

add this bit of syntax to the trait

# Self Types

We want any class that extends **`DormResident`** to also be a **`Person`**

```
trait DormResident { self: Person =>
  val livesOnCampus = true
}
```

add this bit of syntax to the trait

# Self Types

We want any class that extends **DormResident** to also be a **Person**

Now, try to create a class that is-a **DormResident**, but not a **Person**

```
scala> class Student(f:String, l:String) extends DormResident
<console>:14: error: illegal inheritance;
 self-type Student does not conform to DormResident's selftype DormResident with Person
       class Student(f:String, l:String) extends DormResident
                                                 ^
```

# Self Types

## Now, try to create a class that is-a `DormResident`, but not a `Person`

```
scala> class Student(f:String, l:String) extends DormResident
<console>:14: error: illegal inheritance;
 self-type Student does not conform to DormResident's selftype DormResident with Person
       class Student(f:String, l:String) extends DormResident
                                                 ^
```

## Scala will not allow it

# Self Types

## Now, try to create a class that is-a `DormResident`, but not a `Person`

```
scala> class Student(f:String, l:String) extends DormResident
<console>:14: error: illegal inheritance;
 self-type Student does not conform to DormResident's selftype DormResident with Person
       class Student(f:String, l:String) extends DormResident
                                                 ^
```

## Scala will not allow it

# Self Types

## Now, try to create a class that is-a `DormResident`, but not a `Person`

```
scala> class Student(f:String, l:String) extends DormResident
<console>:14: error: illegal inheritance;
 self-type Student does not conform to DormResident's selftype DormResident with Person
       class Student(f:String, l:String) extends DormResident
                                                 ^
```

## Scala will not allow it

# Self Types

## Make sure the class has both types, and all will be fine!

```
class Student(f:String, l:String)
extends Person(f,l) with DormResident;
```

# Self Types

Make sure the class has both types, and all will be fine!

```
class Student(f:String, l:String)
extends Person(f,l) with DormResident;
```

The self-type constraint in the trait forces this mixing of types

# Self Types

## Make sure the class has both types, and all will be fine!

```
class Student(f:String, l:String)
extends Person(f,l) with DormResident;
```

## The self-type constraint in the trait forces this mixing of types

Actually, Scala eliminates the need for an explicit class that mixes the traits, via

# Dependency Injection

# Dependency Injection

## This refers to the creation, on the fly, of a value that mixes traits

```
val person2 = new Person("V","S") with DormResident
```

# Dependency Injection

This refers to the creation, on the fly, of a value that mixes traits

```
val person2 = new Person("V","S") with DormResident
```

Notice how we skipped creating the intermediate Student class

# Dependency Injection

The term arises because the self-type is "injected" on the fly

i.e. at run-time, not compile-time

# Dependency Injection

two instances of the same class might be created and used in entirely different ways

Since the common ancestor type that "mixes in" the traits now no longer exists