

# CS620c

Static method and static variables and few more things about OO techniques.

.

Joe Duffin

┐

**Eolas Building Room 1.45**

**Email: [Joseph.Duffin@nuim.ie](mailto:Joseph.Duffin@nuim.ie)**

# TwoDShape; the superclass aka the parent class

```
public class TwoDShape
{
    private String colour; // class attribute colour for the colour of the shape.
    /**
     * Constructor for objects of class TwoDShape
     */
    public TwoDShape()
    {
        colour = "Grey"; // set the default colour to "Grey".
    }
    /**
     * Constructor for objects of class TwoDShape
     */
    public TwoDShape(String col)
    {
        colour = col; // initialise the colour attribute using a passed col parameter
    }
    /**
     * setter method for the class instance variable colour
     * @param colourValue the value to set the class instance variable colour to.
     * @return void
     */
    public void setColour(String colourValue)
    {
        colour = colourValue ; // set the class instance variable to the value in the formal parameter val.
    }
    /**
     * getter method for the class instance variable colour
     * @param empty
     * @return the value of the class instance variable colour
     */
    public String getColour()
    {
        return colour; // return the value in the class instance variable colour
    }
}
```

# The Square Class inherits from the TwoDShape Class

```
public class Square extends TwoDShape
{
    private double length; // the length attribute of a square
    /**
     * Constructor for objects of class Square
     */
    public Square(double num)
    {
        length = num; // set length to the passed value of num
    }
    /**
     * Square constructor :
     * @param len the length of the side of the square
     */
    public Square(double len, String colourValue) //Notice there is NO return type for a class constructor.
    {
        super(colourValue); // use the constructor in the parent (super class)
        length = len; // set the class attribute (variable) radius equal to len
    }
    /**
     * setter method for the class instance variable length
     * @param length the value to set the class instance variable length to.
     */
    public void setLength(double num)
    {
        length = num ; // set the class instance variable to the value in the formal parameter num.
    }
    /**
     * getter method for the class instance variable length
     * @return the value of the class instance variable length
     */
    public double getLength()
    {
        return length; // return the value in the class instance variable length
    }
}
```

---

## Using `super ( ) ;`

The java word **super( )** stands for the constructor of the parent class in an inheritance relationship.

**super( )** on its own invokes the default constructor.

**super (parameter1, parameter2)** calls the parent constructor with two parameters.

It depends on the design of the constructors available in the parent class (the class inherited from)

---

More on **super ( )** and **super;**

The java word **super** is a **synonym** for the **parent class**.

As we saw in the previous slide **super( )** is a call to the **default constructor** of the parent class and it is **implicitly called** in the **constructor of the subclass** as the **first thing** that happens in that constructor.

Using **super(parameters)** i.e. the super constructor with parameters, invokes the **non default constructor** with the appropriate number of parameters .

This must be the first line of **explicit code** inside any constructor method of the **subclass** that wants to explicitly use it.

---

---

## More on **super**( ) and **super**;

The word **super** on its own followed by the “.” operator and a method name from the super (parent) class can also be used to invoke a method from the super (parent) class.

```
super.calculateOrigin(7, 9);
```

This method call could be invoked inside a method by the same name in the subclass. (example not available yet)

This method would then override the method from the super class and by using the method in the super class and adding extra functionality in a method by the same name and signature.

## The key word **this** revisited

In the last lessons we showed how the keyword **this** can be used to disambiguate (distinguish) between a class attribute and a formal parameter which has the same name or same identifier as in the example below:

```
public BasicCircle(int x) {  
  
    //set class attribute x equal to formal parameter x  
  
    this.x = x;  
}
```

You can also use **this ( )** to call the **default constructor** method with a **non default constructor** method defined in a class. You would do this if you wanted to reuse some of the set up or initialising of data from the default constructor in the non default constructor. Just to make you aware of **this ( )** for now but you probably will not see it for the remainder of the module. I will flag it for you if it is introduced.

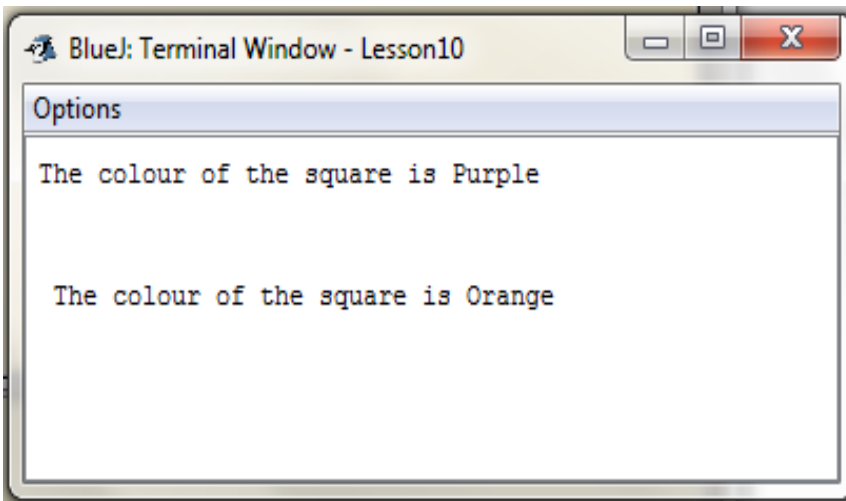
# GeneralShapeTester; creates an object of Square type.

```
public class GeneralShapeTester
{
    public static void main(String [] args)
    {
        // create a square with a length 5 and colour "Purple" (the square constructor here uses the superclass constructor)
        Square mine = new Square(5, "Purple");

        // using the inherited method getColour defined in TwoDShape
        System.out.println("The colour of the square is " + mine.getColour());

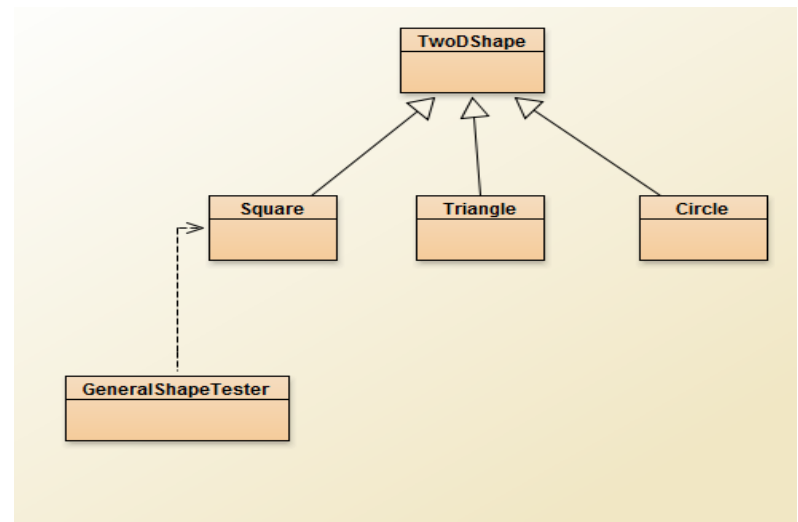
        // using the inherited method setColour defined in TwoDShape
        mine.setColour("Orange");

        // examining the new colour of the square object using the inherited getColour method defined in class TwoDShape
        System.out.println("\n\n The colour of the square is " + mine.getColour());
    }
}
```



```
BlueJ: Terminal Window - Lesson10
Options
The colour of the square is Purple

The colour of the square is Orange
```





---

# Static or Class variables or methods

- Static should really have been called “class”
- Methods and attributes which are Static can have one “version” of them. (E.g. A static variable which counts the number of objects created for a given class).
- We will see later that non-Static items can relate to any number of specific items

# Java wants a main method to run an application.

- The main method is that start for all Java programs so it must be publically accessible hence the **public** access modifier
- The word **static** specifies how to access the main method. With a **non-static** method you must do some extra work prior to accessing it (create an instance of the Class, instantiate a class == an object)
- A static method on the other hand can be accessed immediately without doing any extra work. (Use the Class name followed by “.” followed by the method name.)

# Static variables (Class variables)

- So far all variables were contained within a method. (This includes the main method)
- However, it can be helpful to have a variable accessible to all methods in a class.
  - Without being passed as an argument
  - Variables declared in this way have scope which covers the whole **Class body** { }

---

# Static Variable, Static Methods and Declaring constants in Classes.

It is possible for both variables and methods in a class to be declared **static**. Such variable and methods have special properties.

**Recap:** Each object of a class has its own copy of all of the instance variables defined in the class.

If a class defines an instance variable x (aka field x aka attribute x) and two separate objects a and b are created from that class, then a.x and b.x will be two separate variables.

**It is sometimes useful to have a particular variable in a class shared by all of the objects created from the class.**

---

## Static variable sometimes known as Class variable as there is only one per Class

If a variable is declared static, then **one copy** of that variable will be created the first time a class is loaded by the **JVM**, and that copy **will remain in existence** until the program stops running.

Static variables are also known as Class variables (which is a little confusing, better call them static variables defined in a class).

Any objects instantiated from that class (any objects made with that class template) will **share a single copy of that static variable**.

**Static variables are useful for keeping track of global information, such as the number of objects instantiated from a class (number of objects created) See this week lab 7 assignment.**

# Other use of static variables, creating a class value which is used by instances of a class.

Static variables are also useful for defining single copies of final variables (constants values) which will be shared among all objects of that class (created using a particular class).

For example, the speed of light in a vacuum is  $c = 2.99792458 \times 10^8$  meters per second.

That value could be declared in a class as :

```
static final double c = 2.99792458e8;
```

A single copy of this constant will be created and shared among all of the objects instantiated from the class.

Remember: static makes the variable c a class variable, and final fixes its value and makes it behave as an immutable constant.

The Point class definition which is used in this lesson has pi defined in this way.

```
public static final double pi = 3.141592654;
```

# The duration of a static variable

The duration of a static variable is different from the duration of an instance variable. Instance variables are created each time an object is instantiated and destroyed when the object is destroyed by the garbage collector. In contrast, **static variables are created as soon as the class is loaded into memory, and they persist until program execution ends.**

Although static variables are global in the sense that they are shared among all of the objects instantiated from a class, they still have class scope. They are visible outside of the class only by reference to the class in which they are defined.

**If a static variable is declared public, it may be accessed through any object of that class or directly through the class name using the dot operator.**

```
//Accessing pi through the Class
```

```
System.out.println("The class variable pi " + Point.pi );
```

```
// Accessing pi through and object of the class
```

```
System.out.println("Another way of getting to pi " + p1.pi);
```

Note that static variables in a class can be accessed without first creating an instance of the that class. A good example of this is accessing **PI** defined in the Math class :

```
System.out.println("pi defined in the Math class is:" + Math.PI);
```



# Static methods

**Static** methods are methods that are declared static within a class.

These methods can **directly access the static variables** in the class but cannot **directly access instance variables** . Static methods are also known as **class methods**.

Static methods are commonly used to perform calculations that are independent of any instance data that might be defined in a class.

For example, the method `sqrt ( x )` in the `Math` class calculates the square root of any value passed to it and returns the result.

This method is independent of any data stored in the `Math` class, so it is declared static.

# Static methods continued...

As in the case of static variables, static methods may be accessed by reference to a **class's name without first creating an object of the class**. Thus we are able to use `Math.sqrt( x )` in any program **without** first instantiating an object from the `Math` class.

Look at the example of the public static **calculateDistance** method which we defined in the **Point** class. We can call this method as:

```
double distance = Point.calculateDistance(alpha, delta);
```

A very famous static method is `main`, the starting point of any java application. The `main` method must be static so that it can be executed without first instantiating any object.

**Add a method to the Point class that takes two Point types and uses them to calculate the distance between the two points in 2D space represented by these points.**

```
public static double calculateDistance(Point p1, Point p2){  
  
    double result = 0.0;  
  
    // (x2-x1)^2  
    double a = Math.pow(p2.getX() - p1.getX(), 2);  
  
    // (y2-y1)^2  
    double b = Math.pow(p2.getY() - p1.getY(), 2);  
  
    // square root of ( (x2-x1)^2 + ((y2-y1)^2 ) )  
    result = Math.sqrt( a + b );  
  
    return result;    // return distance between two points.  
  
}
```

**Side Note: How could we get the variable `otherObjRef` to link to or reference (or point to) the same object in memory as referenced by the variable `mySquare`???**

```
// Create a new Square object (with all its baggage)
Square mySquare2 = new Square(13);
```

```
// a variable capable of linking to a Square object.
// at the moment it has a value null in it (it links to
// nothing
Square otherObjRef;
```

```
//otherObjRef now takes on the same value as the value
//as mySquare2 which is a link address to the initially
//created object of type Square.
```

```
otherObjRef = mySquare2;
```

That's why `mySquare2.equals( otherObjRef )` -> evaluates to true

---

Side Note continued: What would happen if we let  
`mySquare2 = otherObjRef;` (Draw box and arrow diagrams 3  
minutes ) ????

This is the scenario below:

```
Square mySquare2 = new Square(13); // (1)
```

```
Square otherObjRef; // (2)
```

```
mySquare2 = otherObjRef; // (3)
```

---

## Conclusion:

We have seen what happens when you use the constructor of a particular class and how it depends on the OO hierarchy (how class definitions are inherited)

We have also examined static variables sometimes (confusingly) known as class variables and we learned how these can be accessed our programs.

We have seen these before but we again spoke about static methods and functionality is written as static methods. Here again there are access rules for objects using these methods and for these methods accessing only static variables.