Chapter 1

Chapter 1

Note that the majority of the actions and exercises can be completed simply by following the instructions in the book.

The following consists of the breaking down of code that you will have seen while doing the exercises and, also, any additional comments deemed necessary.

Project Shapes

Project Shapes

Project Shapes

Note

Ignore the class definition for the **Canvas** class at the moment. It includes elements of Java over and above our current level. Suffice to say that it provides us with a work area upon which to work with our shapes.

Classes

Look at the class definitions for the 3 shapes - Circle, Square & Triangle:

Circle

Class definition for a **Circle**

Attributes

```
private int diameter;
private int xPosition;
private int yPosition;
private String color;
private boolean isVisible;
```

diameter - determines the size of the circle
xPosition & yPosition - determines the placement of the circle
color - determines the colour of the circle
isVisible - determines whether the circle is visible or not

Constructor

```
public Circle()
{
  diameter = 30;
  xPosition = 20;
  yPosition = 60;
  color = "blue";
  isVisible = false;
}
```

The constructor method for **Circle** takes no parameters. It assigns the following values:

diameter	**30**
xPosition	**20**
yPosition	**60**
color	**"blue"**
isVisible	**false**

Note that invoking the **Circle** constructor method **twice** will result in the creation of **2 separate Circle objects** but the **state** of these two objects will be **exactly the same** - they will both have a **diameter** of **30**, an **xPosition** of **20**, **yPosition** of **60**, **colour blue** and be **invisible**.

This will only change when you **invoke the behaviours** or methods of the respective Circle object.

Invoking the behaviours available to you will then **alter the state** of the respective Circle object.

Methods

Note that you are **not** expected to be familiar with the code contained within the methods at this stage.

Benefit

However, you will very likely benefit from attempting to **analyse the respective methods in terms of their associated behaviour**, as has been done with the example below:

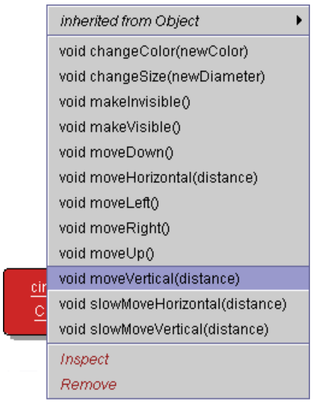
moveVertical

```
public void moveVertical(int distance)
{
    erase();
    yPosition += distance;
    draw();
}
```

The above method, **moveVertical**, moves the circle vertically as follows:

Entering a positive value i.e. 20 moves the circle downwards whilst entering a negative value i.e. -20 moves the circle upwards.

As can be seen from the method signature there is **no return type** - **void** is specified - and the method expects to be passed a **parameter** of type **int**.


inherited from Object
void changeColor(newColor)
void changeSize(newDiameter)
void makeInvisible()
void makeVisible()
void moveDown()
void moveHorizontal(distance)
void moveLeft()
void moveRight()
void moveUp()
void moveVertical(distance)
void slowMoveHorizontal(distance)
void slowMoveVertical(distance)
Inspect
Remove
cir
C

Note that the first line of the body of the method is a call to another method that has been defined within the Circle class - **erase( )**

The **erase( )** method does as its name suggests - it erases the object from its current position before moving it to its new position. The value of **yPosition** is then increased by the amount of the **distance** and a call to yet another method - the **draw( )** method - is made.

Have a look

Have a look at the various methods that have been defined and see what the associated code for a particular behaviour looks like.

If there is a linkage such as that shown in the example above - i.e. one method calls another method - then follow the linkage.

If you do not follow the linkage then you will be left with only a **limited understanding** of how a particular behaviour is performed.

Square

Class definition for a **Square**

Attributes

```
private int size;
private int xPosition;
private int yPosition;
private String color;
private boolean isVisible;
```

size - determines the size of the square
xPosition & yPosition - determines the placement of the square
color - determines the colour of the square

isVisible - determines whether the square is visible or not

Constructor

```
public Square()
{
    size = 30;
    xPosition = 60;
    yPosition = 50;
    color = "red";
    isVisible = false;
}
```

As with the constructor for **Circle**, **no parameters** are passed.

The object is created with the values shown above and, as such, two or more instances of a Square object will be created with the **exact same state** i.e. **size 30**, **xPosition 60** and so on.

Again as before, the state of two or more Square objects will remain the same until you **invoke the behaviours** or methods of the respective Square object.

Methods

As with the **Circle**, you are not expected to be able to identify exactly what is happening in the code. However, I would stress that it is again worth looking at the respective methods to see how they are coded.

The following method is more complex than I would expect you to feel happy with at this moment in time. However, by breaking it down into constituent parts, we can simplify the workings of the method considerably.

The method is passed a parameter - **distance** - that is used to **move the object horizontally.**

Local Variable

The first line of the method body declares a **local variable** of type **int** called **delta**. Note that this particular method moves the object not only horizontally but also **slowly**. This is achieved by moving the object **one pixel at a time** and this is where the local variable **delta** finds its use.

Conditional Behaviour

If the parameter entered by the user is **negative** i.e. it is **less than 0 (distance < 0)**, then **delta** is **set to -1**.

If this is not the case, the **else** part, then **delta** is **set to 1**.

Look at the last bit of code. You will very likely not be familiar with this piece of code. It is a **loop**. Suffice to say at the moment that the loop starts, **alters the xPosition** of the object, and then **invokes** the **draw( )** method of the class.

Note that the loop will **continue**, **over and over**, until it has moved the object the correct distance. (Loops will be covered in detail shortly.)

However, the use of **delta** should now be obvious - In the course of this continuous looping, the objects **xPosition** is **continuously altered** - **by delta**.

Remember that we set delta to be either **1** or **-1** depending upon the original input from the user.

SlowMove Horizontal

```
public void slowMoveHorizontal(int distance)
{
    int delta;

    if(distance < 0)
    {
        delta = -1;
        distance = -distance;
    }
    else
    {
        delta = 1;
    }

    for(int i = 0; i < distance; i++)
    {
        xPosition += delta;
        draw();
    }
}
```

Continue Looking

As with the **Circle**, have a look through the various methods that have been defined for **Square**.

Triangle

Class definition for a **Triangle**

Attributes

```
private int height;
private int width;
private int xPosition;
private int yPosition;
private String color;
private boolean isVisible;
```

height and **width** - determines the size of the Triangle (height & width)
xPosition & yPosition - determines the placement of the Triangle
color - determines the colour of the Triangle
isVisible - determines whether the Triangle is visible or not

Constructor

```
public Triangle()
{
    height = 30;
    width = 40;
    xPosition = 50;
    yPosition = 15;
    color = "green";
    isVisible = false;
}
```

As with the constructor for **Circle** and **Square**, **no parameters** are passed.

The object is created with the values shown above and, as such, two or more instances of a **Triangleobject** will be created with the **exact same state** i.e. **height** and so on.

Again as before, the state of two or more Triangle objects will remain the same until you **invoke the behaviours** or methods of the respective Triangle object.

Methods

Yet again, I would stress that it is worth having a look through the available behaviours and the respective code for the behaviours.

changeSize

```
public void changeSize(int newHeight, int newWidth)
{
    erase();
    height = newHeight;
    width = newWidth;
    draw();
}
```

The nature of the **changeSize** method should be evident from its name - it enables the user to **change the size of their Triangle object**.

Parameters

To do this, it requires that **two parameters** be provided - for the **height** and **width** of the Triangle object respectively.

The original Triangle object is erased with a call to the **erase( )** method, the **height** and **width** attributes of the object are changed according to the provided values, and, finally, the **draw( )** method is invoked.

Ex 1.9

Exercise 1.9 reads as follows:

Use the shapes from the shapes project to create an image of a house and a sun, similar to that shown in Fig 1.7. While you are doing this, write down what you have to do to achieve this. Could it be done in different ways?

Solution

Note that there are any number of ways that you could complete this, depending upon how and when you invoke the behaviours that are available to you. However, the fundamentals of the problem are as follows:

Requirements

You will require the following objects:

1 Circle
2 Squares
1 Triangle

Visibility

You will need to invoke the **makeVisible** method for each of the objects that you have created.

Colours

Amend the **colours** accordingly i.e. so that the sun is yellow.

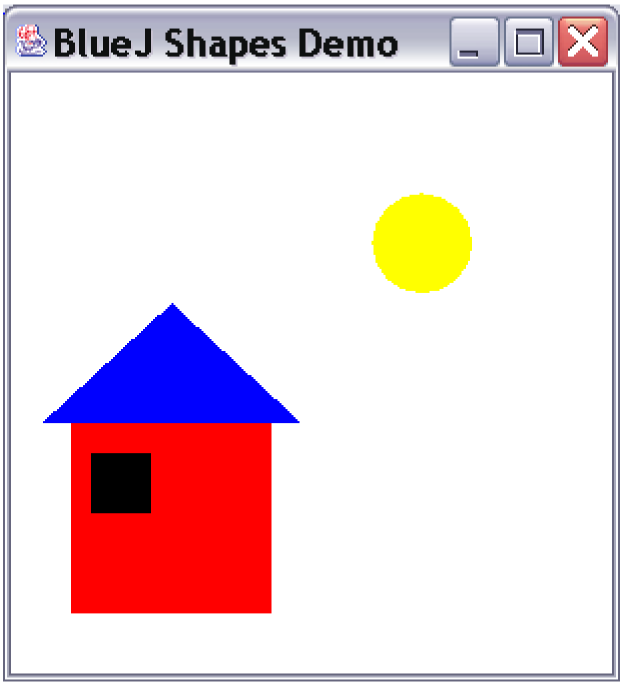
Size

Amend the **sizes** accordingly i.e. so that one square is large enough to form the **walls** of the house whilst the other square is small enough to be used as a **window**.

Move

Amend the **positions** accordingly i.e. so that the sun is in the top right hand corner of the picture, so that the triangle sits in the approximate position for a roof etc etc. (Getting the positionin correct is the worst bit)

Note that you will have to play about for a while to get a respectable looking picture! It is a trial and error process! Here is my effort and I am not saying how long this took!!!



BlueJ Shapes Demo

Project Picture

Project Picture

Open the **picture** project.

Note that the initial environment appears as shown below:

Note

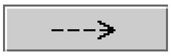
As before, ignore the class definition for the **Canvas** class at the moment.

Why the arrows?

Why the arrows?

The arrows denote **linkages** within the classes. In this particular instance, the **Picture** class **uses** objects of type **Square**, **Circle** and **triangle**.

Note that a use relationship/dependency can be inserted using the button shown below. Hovering over the button with your mouse causes an appropriate 'tool tip' (shown right) to display.



Create an instance of class **Picture** and invoke its **draw** method.

Also, try out the **setBlackAndWhite** and **setColor** methods.

Ex 1.11

How do you think the **Picture** class draws the picture?

It should hopefully be apparent to you that, at the lowest level, the picture is merely **a composite of squares, triangles and circles**.

Specifically, the picture is a composite of **2 squares**, a **triangle**, and a **circle** - the exact same composite that you had by the time you had finished working with the **Shapes project**.

In terms of Java, we can consider that the picture object is composed of two square objects - one for the walls and one for the window - a triangle object, and a circle object.

Object Composition

This is **object composition**, where one object is **composed of other objects**.

Important

To reiterate:

"The important point here is: objects can create other objects and they can call each other's methods."[1]

Picture class

Look at the code for the Picture Class:

First we have a comment:

```
/**
 * This class represents a simple picture. You can draw the picture using
 * the draw method. But wait, there's more: being an electronic picture, it
 * can be changed. You can set it to black-and-white display and back to
 * colors (only after it's been drawn, of course).
 *
 * This class was written as an early example for teaching Java with BlueJ.
 *
 * @author  Michael Kolling and David J. Barnes
```

[1] Barnes, D.J. & Kolling, M; Objects First With Java, A Practical Introduction Using BlueJ; p11

```
 * @version 1.1  (24 May 2001)
 */
```

Then we have the beginning of the class definition for the **Picture** class.

```
public class Picture
{
```

This is followed by the **attributes / fields / instance variables** for the Picture class.

```
    private Square wall;
    private Square window;
    private Triangle roof;
    private Circle sun;
```

The attributes:

Note that the instance variables for the **Picture** class are as follows (remember that the Picture class deals with object composition, whereby one object - the Picture object - is composed of other objects - 2 Square objects, a Triangle object and a Circle object):

One **Square** object with the reference **wall**.
One **Square** object with the reference **window**.
One **Triangle** object with the reference **roof**.
One **Circle** object with the reference **sun**.

The attributes are followed by a **comment** denoting the start of the constructor method for the Picture class.

```
/**
     * Constructor for objects of class Picture
 */
```

This is followed by the **constructor** for the Picture class:

```
public Picture()
{
    // nothing to do...
    //instance variables are automatically set to null
}
```

As can be seen, in this instance the constructor method does nothing. It is not until the user invokes the **draw** method that things start to happen as can be seen from the following code:

First comes the comment to denote the start of the draw method:

```
/**
 * Draw this picture.
 */
```

This is followed by the method **signature**. As can be seen, there are **no parameters** passed to the **draw** method.

```
public void draw()
{
```

Note that this method makes use of the object references that have been defined - **wall**, **window**, **roof** and **sun**.

The method invokes methods defined in class **Square**, **Circle** and **Triangle** so that the **state** of the objects can be **altered** - into something that resembles the desired picture.

The code itself is similar in terms of what it does - it creates the **actual objects** that we created **references** for in the class attributes - making use of the **new** keyword.

For example, the line **`private Square wall;`** created a reference called **wall** to an object of type **Square**. However, it is not until the line **`wall = new Square();`** that an actual object of type **Square** is created.

```
wall = new Square();
wall.moveVertical(80);
wall.changeSize(100);
wall.makeVisible();
```

These above four lines perform the following actions:

The reference - **wall** - that we had created for objects of type **Square** is made to point to a new Square object.

The three following lines then **invoke various methods** that are available to objects of type **Square** - **moveVertical**, **changeSize** and **makeVisible**, passing appropriate parameters where required.

The end result is to construct the **wall** needed for the house in our **Picture** object.

Note that this demonstrates an object, in this case a Picture object, invoking the methods of another object, the Square object that has been used as the wall of the house in the Picture.

```
window = new Square();
window.changeColor("black");
window.moveHorizontal(20);
window.moveVertical(100);
window.makeVisible();
```

The above five lines perform essentially the same task. As with the wall reference variable, the reference variable **window** is made to point to a new **Square** object that is created.

Again as before, the various methods available to objects of type square are invoked and appropriate parameters are passed where required.

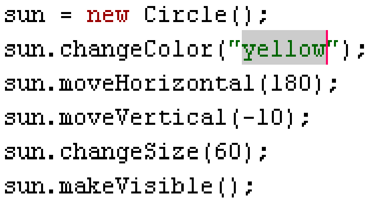
The end result is to construct the **window** needed for the house in our **Picture** object.

```
roof = new Triangle();
roof.changeSize(50, 140);
roof.moveHorizontal(60);
roof.moveVertical(70);
roof.makeVisible();

sun = new Circle();
sun.changeColor("yellow");
sun.moveHorizontal(180);
sun.moveVertical(-10);
sun.changeSize(60);
sun.makeVisible();
```

The above code performs the same task for the **sun** and **roof**, invoking the available methods on the **Triangle** and **Circle** objects respectively.

The end result is the construction of the final components required for our **Picture** object.

The code to the right, for example, determines the **colour** of the sun object in the Picture - **yellow**.



```
sun = new Circle();
sun.changeColor("yellow");
sun.moveHorizontal(180);
sun.moveVertical(-10);
sun.changeSize(60);
sun.makeVisible();
```

Finally, we have the end of the **draw** method, denoted by the terminating **parentheses }**

```
}
```

The following method, **setBlackAndWhite,** enables the user to change the colour picture to one that is **black and white only**.

The **wall** reference is checked to see if it is **not null**. If it **is** null then this means that it **does not currently point to a particular Square object**. In other words, the **draw** method that we have just looked at will **not** have been called. If it had been called then the wall reference variable would have been pointed to a new **Square** object.

If there is no picture, then there is no point in changing it to black and white!

Consider what is being tested by the conditional behaviour here… if the wall reference variable is **not null**.. in other words, if it **does** point to a particular Square object, then the Picture **has been painted**, so it does make sense to perform the actions of the **setBlackAndWhite** method and change the colour of the objects in the Picture.

As long as the condition evaluates to **true**, then the **actions will be performed**. In this case, the actions are calls to invoke the **changeColor** method available to **Square**, **Triangle** and **Circle** objects.

As can be seen, the String parameter **"black"** (or **"white"** for the window) is passed to the **changeColor** method in each case.

```
/**
 * Change this picture to black/white display
 */
public void setBlackAndWhite()
{
    if(wall != null)   // only if it's painted already...
    {
        wall.changeColor("black");
        window.changeColor("white");
        roof.changeColor("black");
        sun.changeColor("black");
    }
}
```

The following method reverses the effects of invoking the **setBlackAndWhite** method, returning the objects in the **Picture** to their original colours:

```
/**
 * Change this picture to use color display
 */

public void setColor()
{
    if(wall != null)   // only if it's painted already...
    {
        wall.changeColor("red");
        window.changeColor("black");
        roof.changeColor("green");
        sun.changeColor("yellow");
    }
}
```

As before, the wall object reference is checked to ensure that it is **not null**. As long as it is not null, then the actions are performed - the same object reference and method invokation as before but this time the parameters that are passed are **"red"**, **"black"**, **"green"** and **"yellow"**.

Finally, we have a single closing parentheses **}** to denote the end of the Picture class definition.

```
}
```

Ex 1.14 Adding A Second Sun

Exercise 1.14 reads as follows:

"Add a second sun to the picture."

To do this, pay attention to the attributes close to the top of the class. As has been discussed, the attributes of the Picture class create **four object references** - **wall** and **window**, object references to objects of type **Square**; **roof**, an object reference to an object of type **Triangle**; and **sun**, an object reference to an object of type **Circle**.

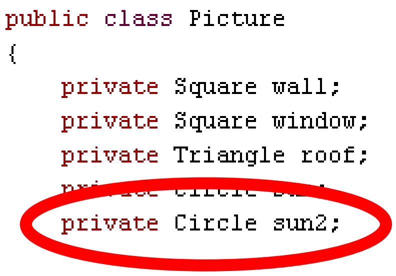
```
private Square wall;
private Square window;
private Triangle roof;
private Circle sun;
```

To add another sun to the Picture requires that we create a further object reference to an object of type **Circle**.

You need to add a line of code to create an object reference for the second sun. For example:

```
private Circle sun2;
```

This is illustrated on the right.

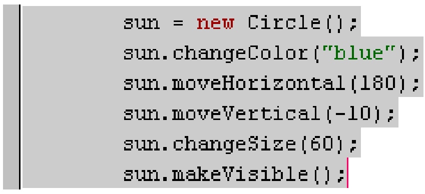


```
public class Picture
{
    private Square wall;
    private Square window;
    private Triangle roof;
    private Circle sun;
    private Circle sun2;
```

Then write the appropriate code for **creating** the second sun.

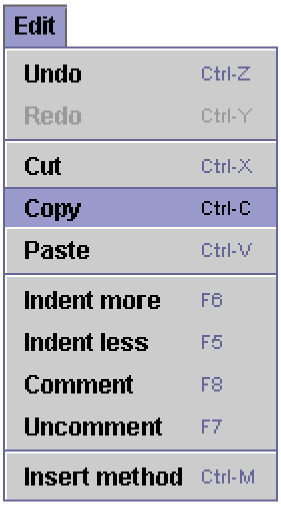
The easiest thing to do would be to find the code that is used in the creation of the first sun.

Select the code, as illustrated on the right...



```
        sun = new Circle();
        sun.changeColor("blue");
        sun.moveHorizontal(180);
        sun.moveVertical(-10);
        sun.changeSize(60);
        sun.makeVisible();
```

Copy the code…



Paste the code beneath the code that you have just copied.

You will need to change a few things.

Changes

Firstly, you will need to amend all of the **sun** references to **sun2** as this is the reference name that you are using to refer to your second sun object.

Note that this is the **easiest mistake to make** - People generally forget that the methods invoked must be invoked for the correct object - sun2 - not sun, the original object.

You will then need to make appropriate changes to the method calls.

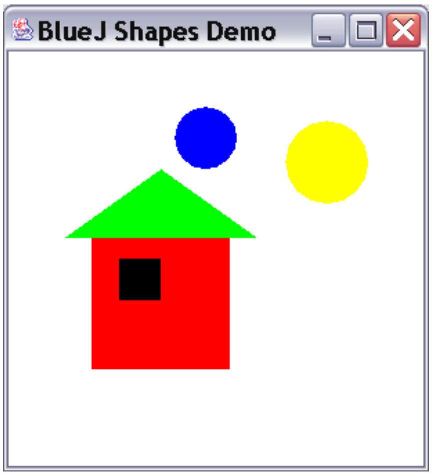
The changes I made are shown below. These are merely suggested changes. You can select your own parameters if you want.

```
sun2 = new Circle();
sun2.changeColor("blue");
sun2.moveHorizontal(100);
sun2.moveVertical(-20);
sun2.changeSize(45);
sun2.makeVisible();
```

Remember to **Compile** the class now that you have amended it.

Create **a new instance** of a Picture object and then call its **draw** method.

Your picture should appear similar to the one on the right. Note that you will now have two separate suns.



Ex 1.15 **Challenge Exercise 1.15 reads as follows:**

Add a sunset to the single-sun version of **Picture**.

That is: make the sun go down slowly.

Remember: The circle has a method **slowMoveVertical** that you can use to do this.

Solution: My suggested solution for this exercise is as follows:

The code below illustrates how the sunset could be achieved.

The line: `sun.slowMoveVertical(250);`

has been added at the bottom of the **Picture** class's **draw** method.

```
sun = new Circle();
sun.changeColor("yellow");
sun.moveHorizontal(180);
sun.moveVertical(-10);
sun.changeSize(60);
sun.makeVisible();
sun.slowMoveVertical(250);
```

Invoking the Circle's **slowMoverVertical** method with a parameter of **250** (suggested value only) ensures that the object descends right off the bottom of the canvas, thereby recreating a sunset effect.

Ex 1.16

Challenge Exercise 1.16 reads as follows:

Challenge Exercise: If you added your sunset to the end of the **draw** method (so that the sun goes down automatically when the picture is drawn), change this now. We want the sunset in a separate method, so that we can call **draw** and see the picture with the sun up, and then call **sunset** (a separate method) to make the sun go down.

The required change is easy enough to make.

We can remove the line

```
sun.slowMoveVertical(250);
```

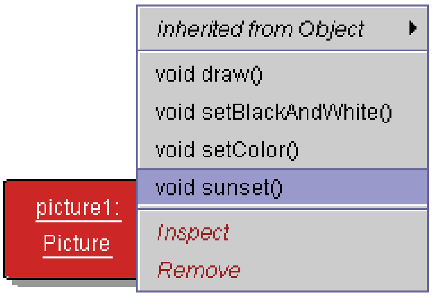
from the draw method and, instead, construct a sunset method that contains the line. The effect of this will be to give the user control over the sunset.

This is illustrated below:

```
/**
 * sunset method
 * Method to recreate the setting of the sun
 */
public void sunset()
{
    sun.slowMoveVertical(250);
}
```

Compile the class and then create a new instance of a Picture object. Call the **draw** method for the object. The picture should be drawn as before but without the automatic sunset.

Right-click on your Picture object. Your additional method should now appear as an available method / behaviour for you to select:



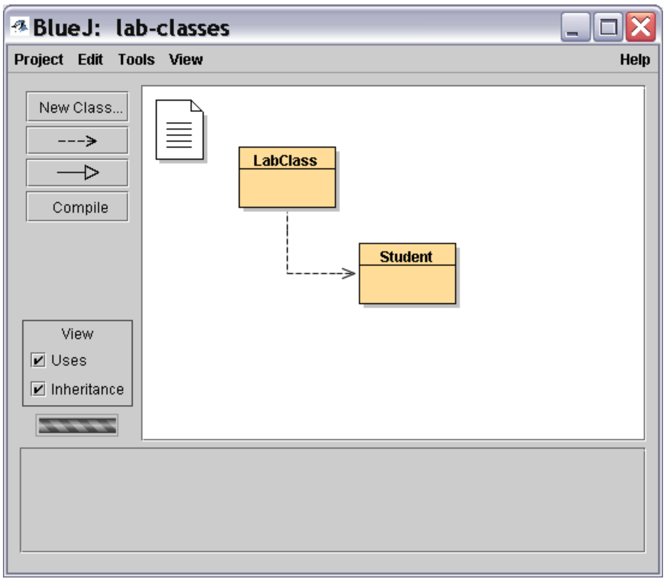
You should now see your circle move horizontally down the canvas.

Voila, a sunset!

Project Lab Classes

Project Lab Classes

Open the **lab-class** project. The initial environment appears as shown below: Note that this again indicates a use/dependency relationship between the classes: **LabClass - Student**.

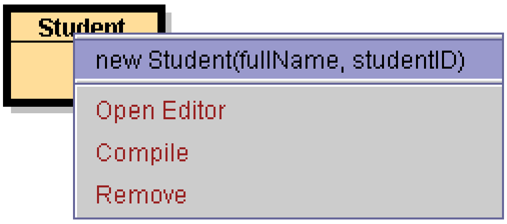


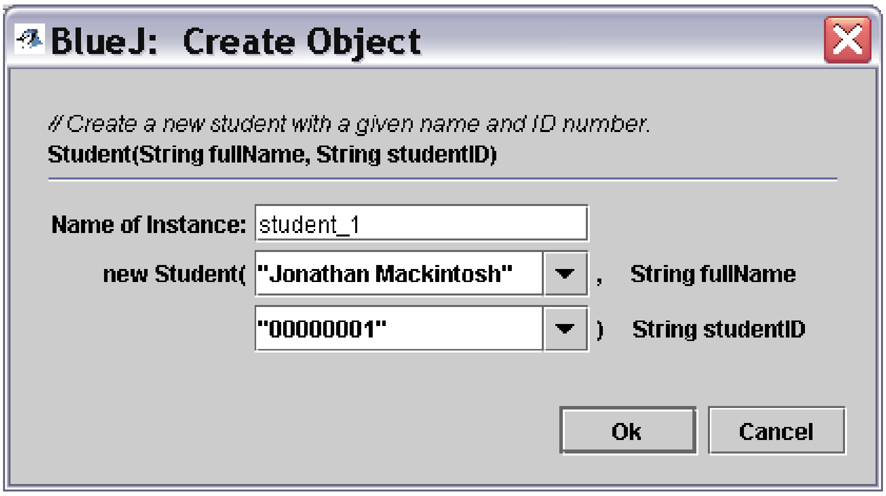
Ex 1.17

When you create an object of class **Student** you will see that you are prompted not only for a name of the instance, but also for parameters relating to the **name** and an **id** for the student object.

Fill them in before clicking **OK**.

Remember that parameters of type **String** must be written in double quotes i.e. **"Jonathan J Mackintosh"**, **"00000001"**





BlueJ: Create Object

// Create a new student with a given name and ID number.
Student(String fullName, String studentID)

Name of Instance: student_1

new Student("Jonathan Mackintosh" , String fullName

"00000001") String studentID

Ex 1.18

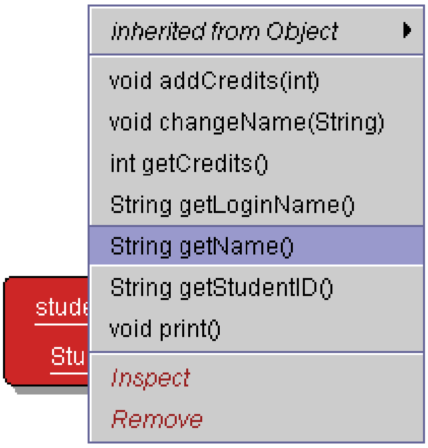
Create some Student objects. Call the **getName** method on each object. Expain what is happening.

Solution

When you create instances of Student objects you are creating **individual objects** using the Student class definition.

Each time you create a Student object you will be prompted for values for the **fullName** and **studentID** attributes.

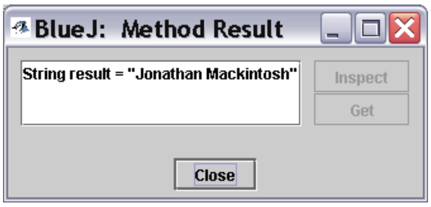
The values provided will be **specific to the individual students** and will, thus, result in the creation of Student objects with **different state**.



When you invoke the **getName** method on each of the Student objects you are requesting the **current value** of the **fullName** attribute for **that particular instance of a Student object** and, you would, therefore, expect the name that is returned to be **different for each Student**.

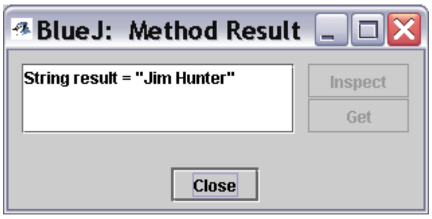
This is demonstrated on the right.

If I had created **two** Student objects, had provided the values **"Jonathan J Mackintosh"** and **"Jim Hunter"** and had then invoked the **getName** method for each of these objects, then I would expect to see the dialog boxes shown to the right:



BlueJ: Method Result

String result = "Jonathan Mackintosh"

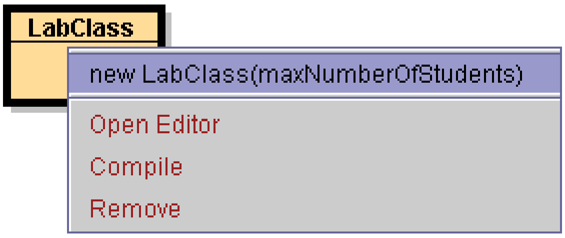


BlueJ: Method Result

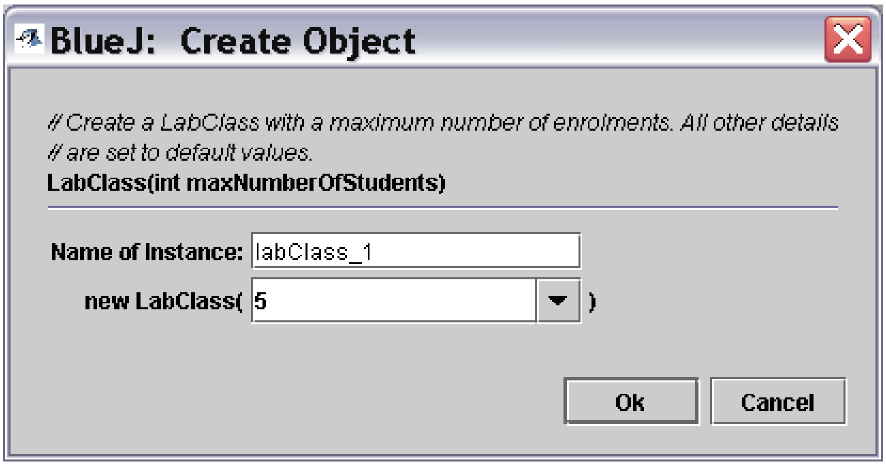
String result = "Jim Hunter"

Ex 1.19

Create an object of class **LabClass**. As the signature indicates, you need to specify the **maximum number of students in that class** (an integer).



Insert, for example, **5** as the size for the new **LabClass**.



// Create a LabClass with a maximum number of enrolments. All other details
// are set to default values.
LabClass(int maxNumberOfStudents)

Name of Instance: labClass_1

new LabClass(5)

Click **Ok**

Note Note that you could call the **LabClass CS5036** for example.

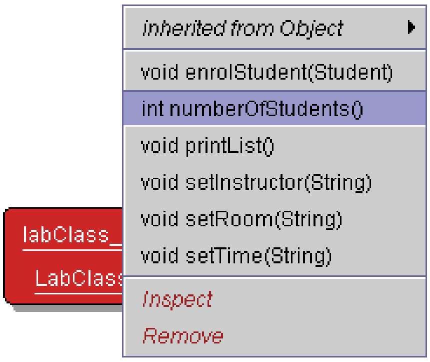
You could then also create **other instances of LabClass** with **names that reflect your other course codes**.

This may help you to understand how we can create **many instances** from **one class definition** if you are finding the concept at all sticky.

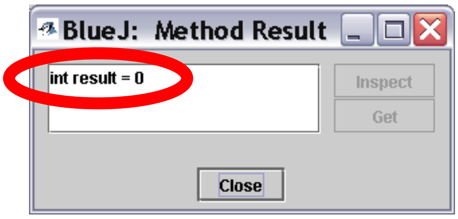
Ex 1.20 Call the **numberOfStudents** method of that class. What does it do?

The **numberOfStudents** method will return a value that indicates **how many Students have been enrolled on that class**.

As you would expect, until you have explicitly enrolled some students on to the LabClass, then the result is **0**, representing the fact that we currently have **no students**.



This is demonstrated to the right:



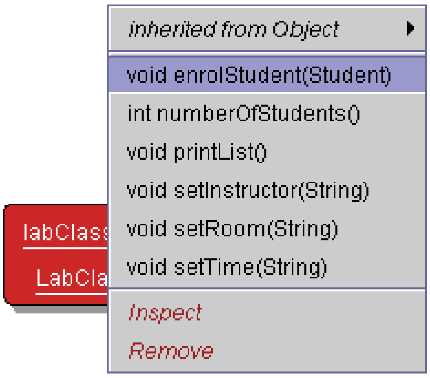
int result = 0

Ex 1.21

Look at the signature of the **enrollStudent** method (below).

You will notice that the type of the expected parameter is **Student**.

Make sure that you have two or three students and a **LabClass** object on the object bench, and then call the **enrollStudent** method of the **LabClass** object.

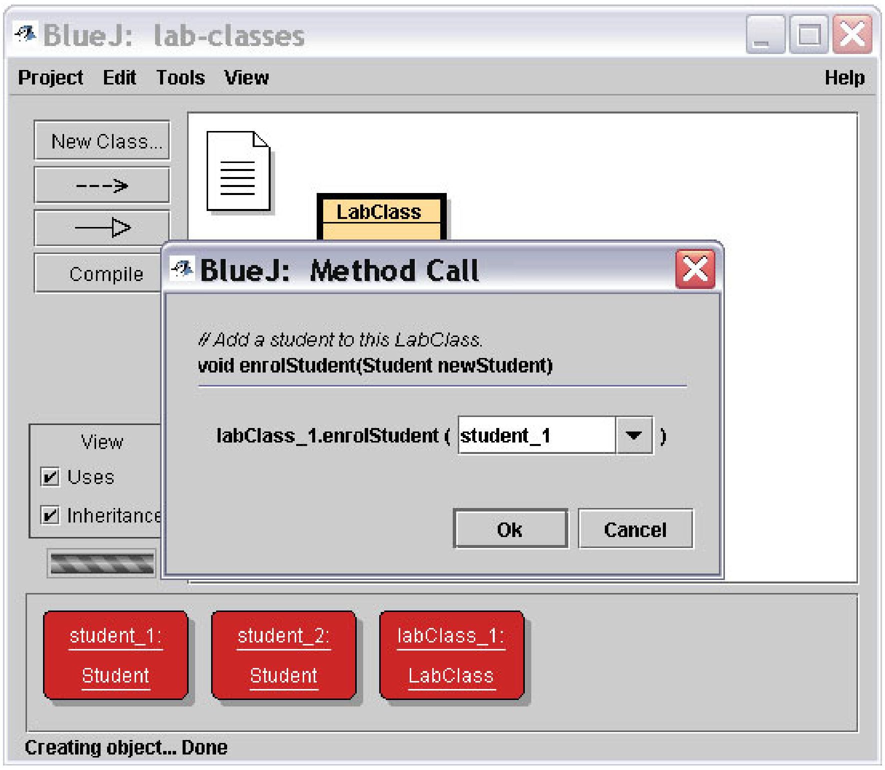


enrollStudent

```
public void enrolStudent(Student newStudent)
```

With the input cursor in the dialog entry field, click on one of the student objects - this enters the name of the student object into the parameter field of the **enrollStudent** method (Figure 1.8). Click OK, and you will have added the student to the **LabClass**.

Add one or more other students as well.

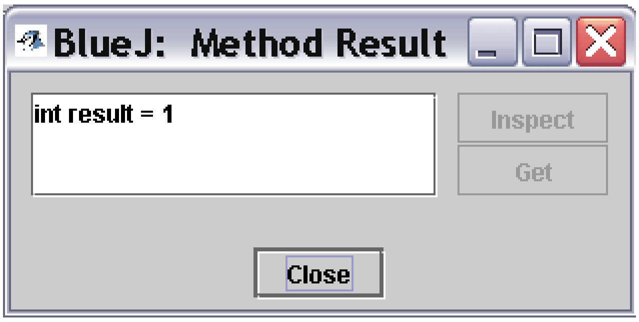


// Add a student to this LabClass.
void enrolStudent(Student newStudent)

labClass_1.enrolStudent (student_1)

Creating object... Done

Note that, if you were to call the **numberOfStudents** method for the lab class now it would **return a different result**, as shown below:

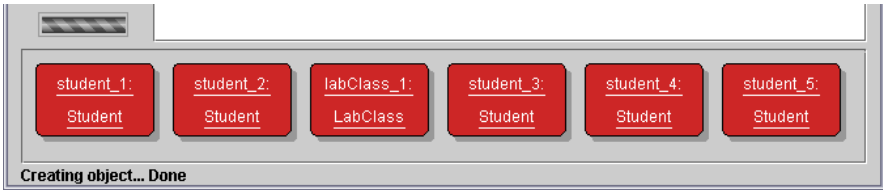


int result = 1

Ex 1.23 Create three students with the following details:

Name	Student ID	Credits
Snow White	100234	24
Lisa Simpson	122044	56
Charlie Brown	12003P	6

The object bench should appear as follows:

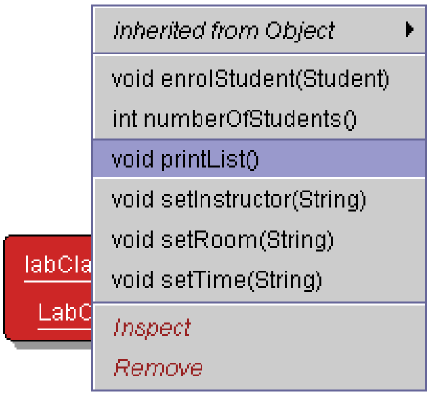


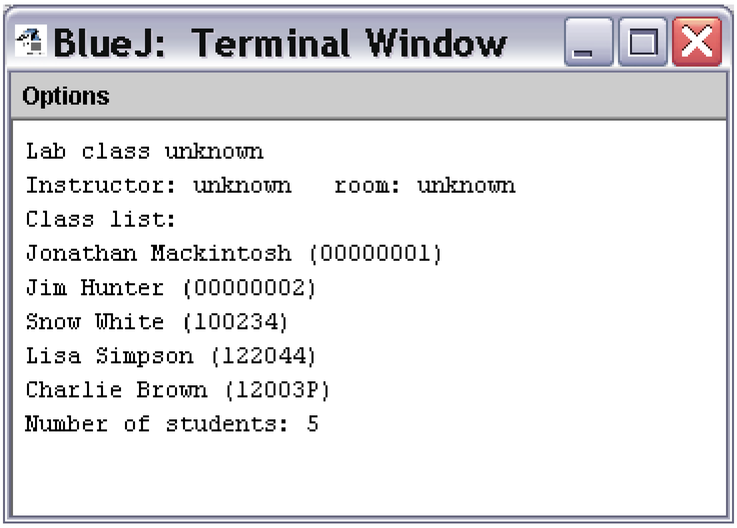
Creating object... Done

Then enter all three into a lab and print a list to the screen.

Invoke the **printList** method.

You should have something similar to what is shown below:





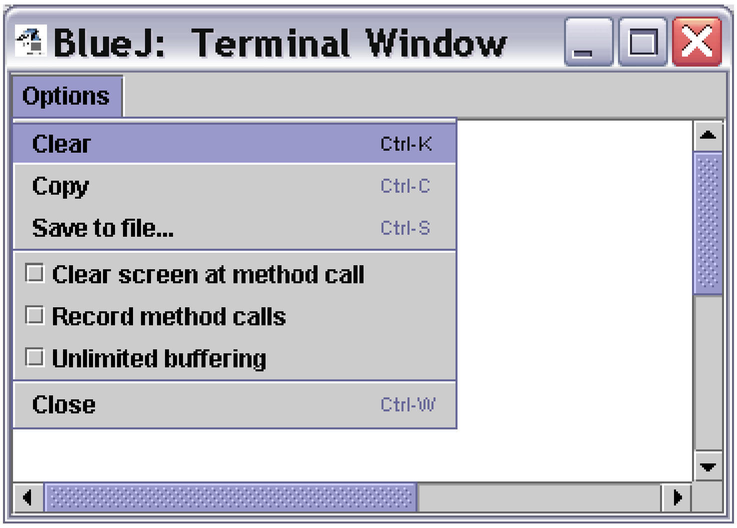
```
Lab class unknown
Instructor: unknown   room: unknown
Class list:
Jonathan Mackintosh (00000001)
Jim Hunter (00000002)
Snow White (100234)
Lisa Simpson (122044)
Charlie Brown (12003P)
Number of students: 5
```

Note: Clearing the Terminal Window

Note that you will likely wish to **clear the Terminal Window** once you have checked that the window displays what you expected.

If you do not clear the window, the **displays will run after each other** which may cause some confusion i.e. you will end up with results that may no longer be applicable.

Simply closing the terminal window will **not** clear the contents of the window. To clear it, you must explicitly select the **Clear** option from the **Options** menu as shown below:

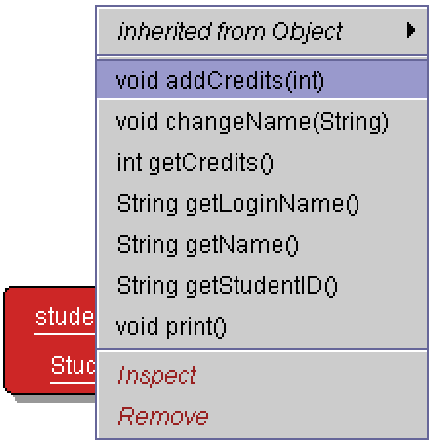


addCredits

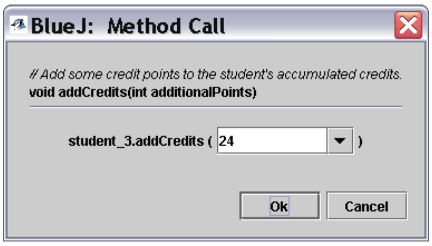
Note that 1.23 asks you to create three students - **Snow White**, **Lisa Simpson & Charlie Brown** - with 'the following details'…

The provided details include **credits** i.e. Snow White has 24 credits, Lisa Simpson 56 etc. etc.

However, you are **not specifically instructed to allocate these credits to the students**.



You can allocate the credits by right clicking on the appropriate student object and then selecting the **addCredits** method.



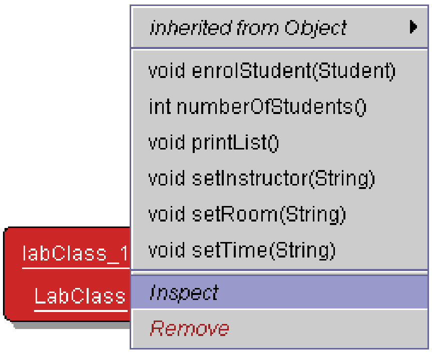
```
// Add some credit points to the student's accumulated credits.
void addCredits(int additionalPoints)

student_3.addCredits ( 24 )
```

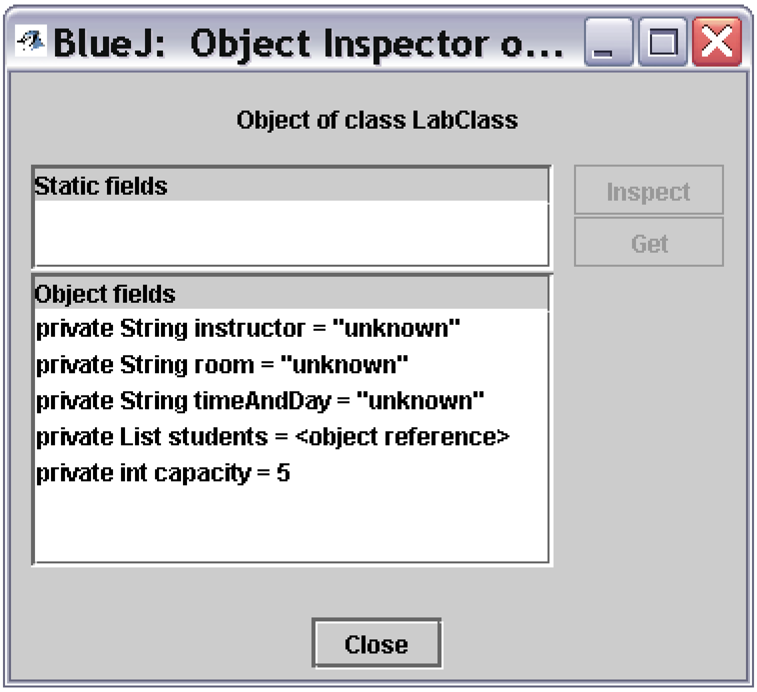
Ex 1.24

Use the **inspector** on a **LabClass** object to discover what fields it has.

Right click on the object that you wish to inspect and then select the **Inspect** option from the menu that appears (shown right).

This results in the display shown below. As you can see, **Inspect** enables you to 'inspect' **all of the attributes** of an object.





Object of class LabClass

Static fields

Object fields

private String instructor = "unknown"

private String room = "unknown"

private String timeAndDay = "unknown"

private List students = <object reference>

private int capacity = 5

Ex 1.25 Set the **instructor**, **room and time for a lab**, and **print the list** to the terminal window to check that these new details appear.

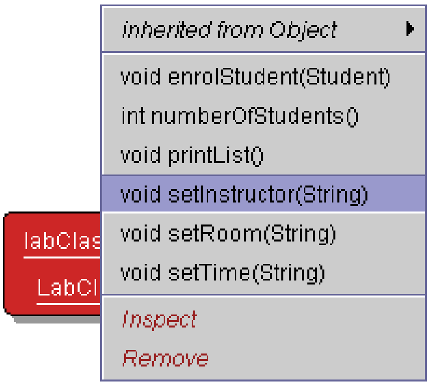
Right-click on the lab class object.

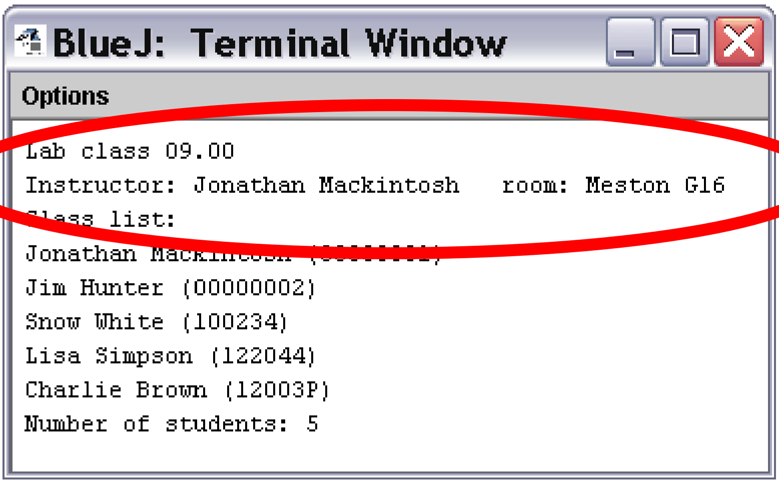
You should see the required methods **setInstructor**, **setRoom** and **setTime**.

Add appropriate details as prompted when you call each of the methods i.e.

"**Jonathan Mackintosh**"
"**Meston G16**"
"**09.00**"

Then print the list to the terminal as instructed.





```
Lab class 09.00
Instructor: Jonathan Mackintosh   room: Meston G16
Class list:
Jonathan Mackintosh (00000001)
Jim Hunter (00000002)
Snow White (100234)
Lisa Simpson (122044)
Charlie Brown (12003P)
Number of students: 5
```

Ex 1.26

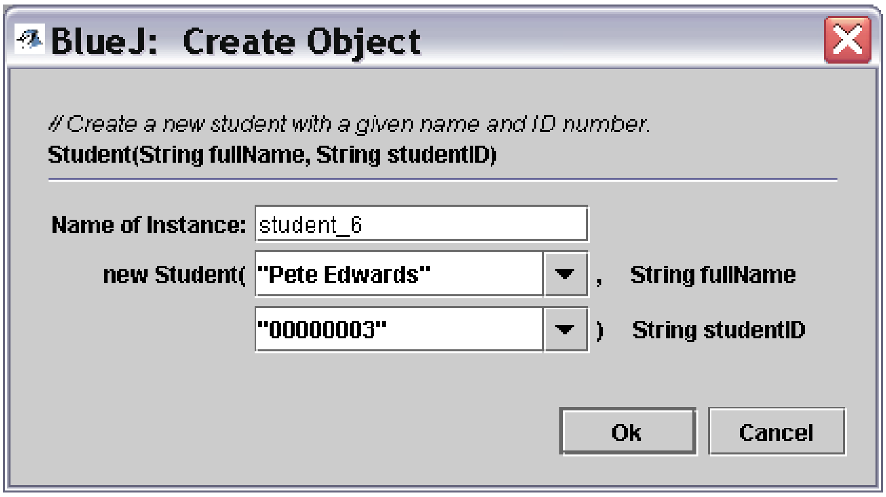
In this chapter we have mentioned the data types **int** and **String**. Java has more predefined data types. Find out what they are and what they are used for. To do this, you can check Appendix B, or look it up in another Java book or in an online Java manual. One such manual is at:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

Additional [non-book]

Additional: Create **one further student object** so that there are **6** student objects on the object bench (if you did not use 5 as the max size for the lab class then add enough student objects so that you can attempt to enroll too many students).

For example, we could create an additional student with the details provided below:

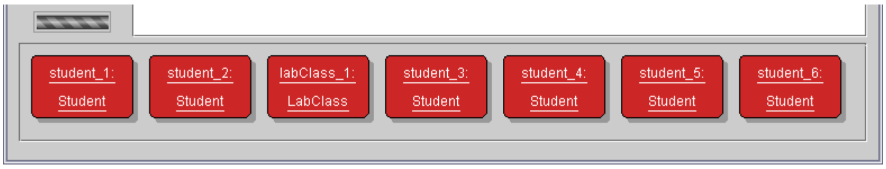


// Create a new student with a given name and ID number.
Student(String fullName, String studentID)

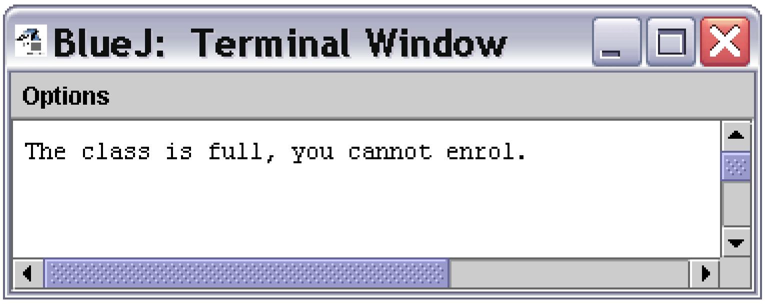
Name of Instance: student_6

new Student("Pete Edwards" , String fullName
"00000003") String studentID

As can be seen from the illustration of the object bench shown below, we now have **too many - 6 - students** for the **capacity of our lab class - 5**.



If we attempt to **add the sixth student** to the lab class, we receive the message shown below:



```
The class is full, you cannot enrol.
```

Calling the **numberOfStudents** method for the lab class shows that there are 5 students enrolled for the class. Thus, our attempt to enroll an additional student has been well and truly rejected.

This complies with the **sensible behaviour** that we would expect.

Look

Look at the code for both the **Student** class and the **LabClass**.

As before, attempt to identify the code behind certain behaviours.

Note, however, that you may not be able to fully understand the code at this stage. The point of this exercise is more to do with familiarity with objects working together.