OOPJAV – Object-Oriented Programming in Java

Seminar 4 - Math Class, Method call, Abstract methods, Basic GUI, Events

Well done on reaching the top of the first steep climb! Things will now level off a bit as we trek across a gentler slope, consolidating what we have learned (and picking a few new things up on the way) before commencing the next ascent. We will start with a quick overview of the arithmetic operators introduced somewhat casually last week and then look at the Math class which contains some useful arithmetic. We will then revisit method calls just to make sure that we all understand the mechanism. Next, we will look at some new subject matter, starting with type conversions, to allow us to do *mixed mode arithmetic,* and then we will spend the rest of our time having a more detailed look at strings (introduced in week 1), and in particular the String class.

I. Overview of Arithmetic Operators

Arithmetic operations are among the most fundamental instructions that can be included in a computer program. As we have seen, Java supports all the standard mathematic operations (see **Figure 1**).

Operator	Interpretation
+	Unary plus or addition
-	Unary minus or subtraction
*	Multiplication
/	Division
%	Remainder
++	Incrementation (by 1)
--	Decrementation (by 1)
+=	Add expression
-+	Subtract expression

Figure 1: *Standard mathematic operations*

Note that:

1. In computer programs all formulas must be written on one line. For example, 10/2 (ten divided by two).

2. The % operator will compute the remainder after the division of two integer values. For example, 11%2 will give 1.

3. When dividing one integer number (short, int or long) by another, the result is always an integer. For example, 5/2 will be 2 and not 2.5!

4. The ++ and --, incrementation and decrementation operators, are equivalent to add or subtract one to a numeric data item. For example, number++, and number-- are equivalent to:

```
number = number + 1;
number = number - 1;
```

5. The += and -= are used to add/subtract an arithmetic expression to the value of a data item. For example:

```
number = number + (6 / 2);
number = number - (6 / 2);
```

will be equivalent to:

```
number += 6 / 2;
number -= 6 / 2;
```

6. The ++, --, += and -= are provided as short cuts for frequently required arithmetic operations.

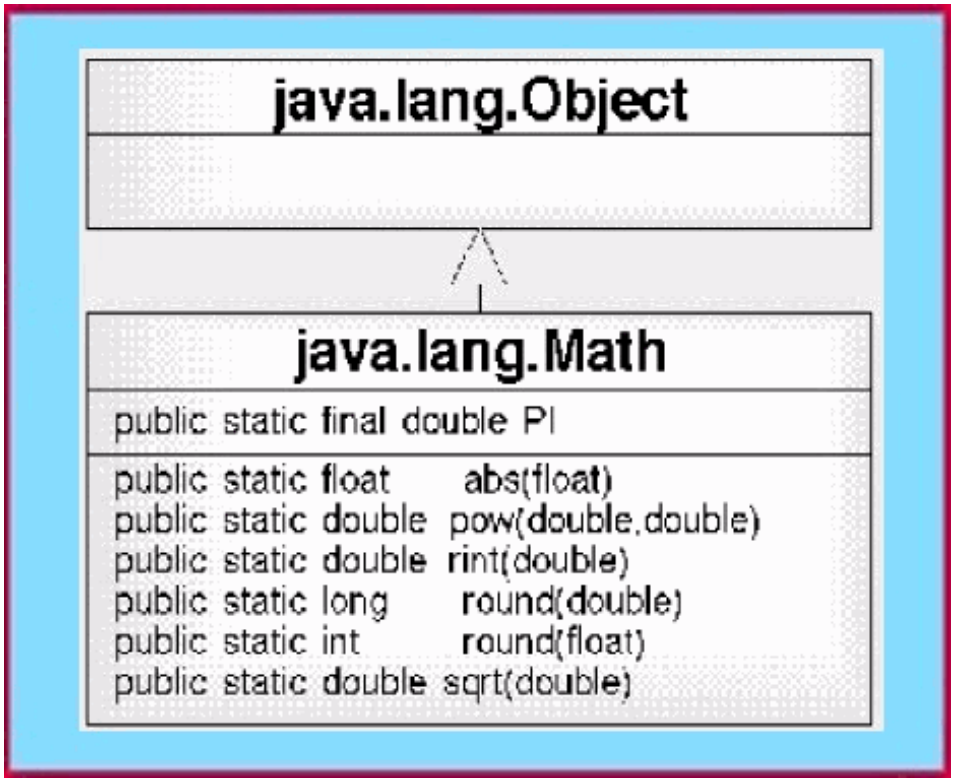

Figure 2: *Java Math class*

More complex arithmetic operations can be found in the Java Math class. A fragment of the Math class is given in **Figure 2**. The class is contained in the java.lang package which is always included in all Java programs. The class includes constants such as Pi and methods such as:

1. abs(float) which returns the *absolute* value of its argument (for example Math.abs(-12.3) will return 12.3. There are *abs* methods for each of the primitive numeric types double, float, int and long.

2. pow(double,double) which returns the value of its first argument raised to the power of the second argument. Note that both arguments must be doubles.

3. sqrt(double) which returns the square root of its argument (a double).

4. round(double) and round(float) which return the closest long integer to the argument and the closest standard integer to the argument respectively.
rint(double) returns the closest "integer" to its argument, but as a data item of type double.

These methods are all class (static) methods and therefore we do not need to create an instance of the class Math to use them --- we simply link them to the class name. For example:

Math.abs(number);

To give a further illustration of the use of arithmetic operators (to supplement those given in the text), and an example of how we access the Math class constant Pi, **Figure 3** gives an example program which is designed to calculate and output the circumference and area of a circle given its radius. The radius is supplied by the user as a double.

```
1  // Circle class
2  // Dr. Y. Jing
3  // 05 December 2007
4  // The University of Liverpool, UK
5
6  class Circle{
7
8  // --- Fields ---
9
10 private double circleRadius;
11 private double circleCircum;
12 private double circleArea;
13
14 // --- Constructors ---
15 public Circle (double newRadius){
16     circleRadius = newRadius;
17     }
18 // --- Methods ---
19 /* calculate circumferences */
20
21 public void calcCircleCircum(){
22     circleCircum = 2.0 * Math.PI*circleRadius;
23     }
24
25 /* Calculate area */
26 public void calcCircleArea(){
27 circleArea = Math.PI*circleRadius*circleRadius;
28 }
29
30 /* Output circumference and area */
31
32 public void outputCircAndArea() {
33     System.out.println("Circumference =" + circleCircum);
34     System.out.println("Area=" + circleArea);
35     }
36 }
```

Figure 3: *Circle class definition*

An appropriate application class that makes use of the Circle class is given in **Figure 4**.

```
1  // CircleApp class
2  // Dr. Y. Jing
3  // 05 Dcember 2007
4  // The University of Liverpool, UK
5
6  import java.io.*;
7
8  class CircleApp{
9
10 // --- Feilds ---
11 // Create Buffered reader class instance
12
13 static InputStreamReader input = new InputStreamReader(System.in);
14 static BufferedReader keyboardInput = new BufferedReader(input);
15
16 // --- Methods ---
17
18 public static void main (String[] args) throws IOException{
19     double radius;
20     // input radius;
21     System.out.print("Input a radius (double):");
22     radius = new Double (keyboardInput.readLine()).doubleValue();
23     // Create new Circle instance
24     Circle newCircle = new Circle(radius);
25
26     // Caculate circumference and area
27
28     newCircle.calcCircleCircum();
29     newCircle.calcCircleArea();
30
31     //Output result
32
33     newCircle.outputCircAndArea();
34     }
35 }
```

Figure 4: *Circle class application program*

Note on trigonometric ratios:

Methods that return the trigonometric ratios (sine, cosine, tangent, cosecant, secant and cotangent) given an appropriate angle are all also available in the Math class. However, it should be noted that the required angle for these methods must be presented in radians and not degrees. To convert from degrees to radians we multiply by:

Math.PI / 180.0

To convert from radians to degree we multiply by:

180.0 / Math.PI

II. Method Calls

The process of calling one method from within another method (as illustrated in section 1) is known as *routine invocation*, where we say that we invoke a method. Routine invocation is activated by a *method call,* which names the method and supplies the *actual parameters* which

are assigned to the *formal parameters* associated with the method. On completion of the invocation, control is returned to the point immediately after the invocation, as illustrated in **Figure 5**.

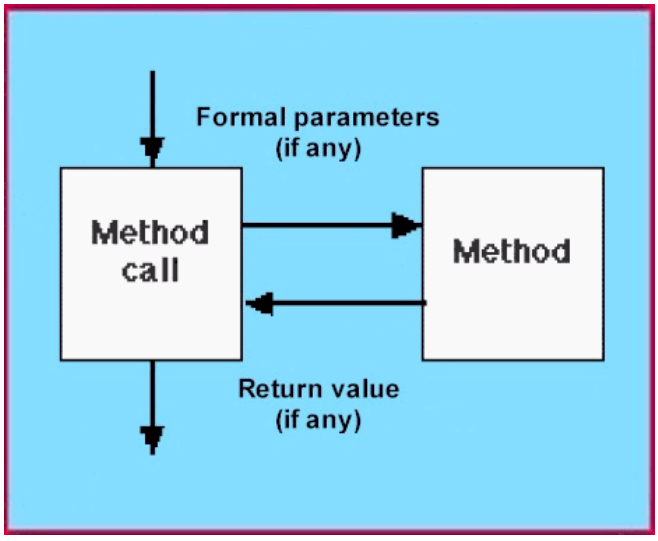

Figure 5: *Flow of control with respect to routine invocation*

If the method we are calling is defined in some other class then we must call it by either:

1. Linking it to the class name if it is a class method

2. Linking it to an instance of the class if it is an instance method

This is illustrated in the code fragment given in **Figure 6**. Here, we define a class with a private instance variable and a public instance method which returns the instance variable multiplied by 3. The code given in **Figure 7** presents an application class that makes use of the class given in **Figure 6**. Two instances are created of the class ExampleClass, instance1 and instance2, each of which calls the treble method in turn. The resulting output will be as follows:

Instance 1 = 6
Instance 2 = 9

```
1  // ExampleClass class
2  // Dr. Y. Jing
3  // 05 Dcember 2007
4  // The University of Liverpool, UK
5
6  class ExampleClass{
7  // --- Fields ---
8  private int value;
9
10 // --- Constructors ---
11
12 public ExampleClass(int num) {
13     value = num;
14 }
15
16 // --- Methods ---
17
18 public int treble () {
19 return (value * 3);
20 }
21 }
```

Figure 6: *Example class*

```
1  // ExampleApplicationClass class
2  // Dr. Y. Jing
3  // 05 Dcember 2007
4  // The University of Liverpool, UK
5
6  class ExampleApplicationClass {
7  // --- Fields ---
8  private int value;
9  // --- Methods ---
10 public static void main (String[] args){
11 // Create two instances of the Example Class
12 ExampleClass instance1 = new ExampleClass(2);
13 ExampleClass instance2 = new ExampleClass(3);
14
15 // Print output
16
17 System.out.println("Instance 1 = " + instance1.treble());
18 System.out.println("Instance 2 = " + instance2.treble());
19 }
20 }
```

Figure 7: *ExampleApplication class with method calls*

The abstract examples given follow on from one another, starting with a very simple example and then building up with subsequent material.

III. Abstract Methods

Consider the class hierarchy given in **Figure 8**. Here, we have a TopClass which has a constructor and a field numX. This class has two subclasses, ClassOne and ClassTwo, both of which have a method called function1 whose signatures (heads) are identical but whose bodies differ - one adds numX to numY and the other multiplies numX by numZ. **Figure 9** presents a class diagram illustrating the class hierarchy associated with these classes.

```
1  // TopClass class
2  // Dr. Y. Jing
3  // 05 Dcember 2007
4  // The University of Liverpool, UK
5
6  class TopClass{
7
8  protected int numX;
9  public TopClass(int value){
10     numX = value;
11 }
12 // Get numX
13
14 public int getNumX(){
15     return (numX);
16 }
17 }
18
19 class ClassOne extends TopClass{
20     public ClassOne(int value){
21         super(value);
22         }
23
24     // Function 1
25     public void function1 (int numY){
26         numX = numX + numY;
27         }
28     }
29 class ClassTwo extends TopClass{
30     public ClassTwo(int value){
31         super(value);
32         }
33
34     // Function 1
35     public void function1 (int numZ){
36         numX = numX * numZ;
37         }
38     }
```

Figure 8: *Class hierarchy*

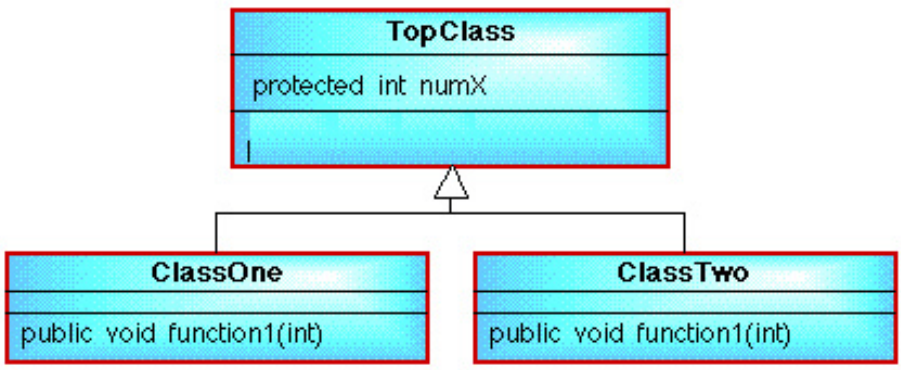

Figure 9: *Class diagram*

An application class which makes use of the classes in the hierarchy is presented in **Figure 10**; some sample output is given below:

$ java AbstractExApp
object1.getNumX() (after object1.function1(2) call) = 14

object2.getNumX() (after object2.function1(2) call) = 12

```
1  // Abstract Example class
2  // Dr. Y. Jing
3  // 05 Dcember 2007
4  // The University of Liverpool, UK
5
6  class AbstractExApp{
7  // Main method
8  public static void main (String[] args){
9  // Create instance of Calssone and ClassTwo
10 ClassOne object1 = new ClassOne(12);
11 ClassTwo object2 = new ClassTwo(6);
12
13 // Output
14 object1.function1(2);
15 System.out.println("object1.getNumX()(after object1.function1(2) call)="
16                    +object1.getNumX()+"\n");
17
18 object2.function1(2);
19 System.out.println("object2.getNumX()(after object2.function1(2) call)="
20                    +object2.getNumX()+"\n");
21 }
22 }
```

Figure 10: *Application*

Given this situation, it would be nice if we could define the head of the function1 method (i.e. its signature) in the TopClass and its implementational detail (i.e. the body) in the sub-classes. Java provides a means of doing this by allowing us to define, in the TopClass, what is known as an a*bstract method*.

An abstract method is one that is defined by its *signature* only, i.e. it has no body. The implementational detail for the method, i.e. the body, is supplied by the sub-classes of the class in which the method is defined according to the nature of each sub-class. We indicate such a method to the compiler using the keyword **abstract**. Therefore, in the above example, we could include in TopClass the method:

abstract public void function1(int dummy);

An abstract method can be thought of as a template or blueprint for a method whose implementational details are contained in the sub-classes of the class in which the abstract method is defined. The implementational detail is included in the normal way, so no changes need to be made to the definition of ClassOne or ClassTwo (see **Figure 1**) in our example. In effect, the abstract method is *overriden* by the full method definitions contained in the sub-classes. Much use is made of abstract methods when creating GUIs.

A. Note on Super Keyword

If we wish to create an instance of a sub-class, we obviously must use the constructor for this class. When invoked, Java will also call the constructors for the superclasses starting with the base-class constructor and working down to the subclass in question. This works fine provided that we only wish to invoke zero-argument (default) constructors. However, in the example given in **Figure 1**, the constructor has an argument! The question is then "how do we get the required argument to this constructor?" The answer is that we use the **super** keyword as illustrated in the above code. The keyword **super** thus allows us to make use of the constructor defined in the

super class, and so (in the above case) we can cause a value to be assigned to the numX data member defined in the super-class.

Note that when using the reserved word **super** it must always be the first statement in the constructor body.

IV. Abstract Classes

An *abstract class* is one that is identified by the keyword **abstract**. An abstract class does not have to contain an abstract method, however, a class that does contain at least one such method is considered to be an abstract class and must therefore be identified using the keyword abstract (otherwise it will not compile). Similarly, an abstract class can contain methods that are not abstract. If we include an abstract method in our TopClass presented in **Figure 8**, TopClass will become an abstract class and must be declared as such. A revised version of the TopClass definition as an abstract class is presented in **Figure 11**.

```
1  // REVISED TopClass class
2  // Dr. Y. Jing
3  // 05 Dcember 2007
4  // The University of Liverpool, UK
5
6  abstract class TopClass{
7
8  protected int numX;
9  public TopClass(int value){
10     numX = value;
11 }
12 // Get numX
13
14 public int getNumX(){
15     return (numX);
16 }
17 // Abstract method function1
18
19 abstract public void function1 (int dummy);
20 }
21
22 class ClassOne extends TopClass{
23     public ClassOne(int value){
24         super(value);
25         }
26
27     // Function 1
28     public void function1 (int numY){
29         numX = numX + numY;
30         }
31     }
32 class ClassTwo extends TopClass{
33     public ClassTwo(int value){
34         super(value);
35         }
36
37     // Function 1
38     public void function1 (int numZ){
39         numX = numX * numZ;
40         }
41     }
```

Figure 11: *Abstract class*

Note that it is not possible to create instances of an abstract class (as there will be no means of implementing the abstract methods it may contain). Similarly, an instance of a subclass of an

abstract class can only be created if any abstract methods in the superclass are overridden in the subclass definition. If not the subclass should also be defined as an abstract class.

It is quite possible to program in Java without resorting to abstract classes; however, they are a feature of the Java API and so any student of Java should at least be aware of the existence of such things! With respect to creating GUIs, their usage, as we will see, is unavoidable.

V. Interfaces

From previous work, we have seen that classes are typically arranged in a class hierarchy (**Figure 12**) such that classes lower down the hierarchy inherit methods and attributes from their superclasses. In Java, inheritance is a one-to-one relationship - a class can only be a subclass of (i.e. inherit from) one superclass, although several subclasses can inherit from the same superclass. Java does not support multiple *inheritance* (**Figure 13**).

However, the effect of multiple-inheritance can be achieved using an *interface* (**Figure 14**). An interface is an abstract class that contains only abstract methods and/or constants. The interface supplies a specification of methods which must be *implemented* by a subclass of the interface, so we say that the subclass *implements* the interface. It is possible for an interface not to contain any methods or constants. Interfaces are another important element when creating GUIs.

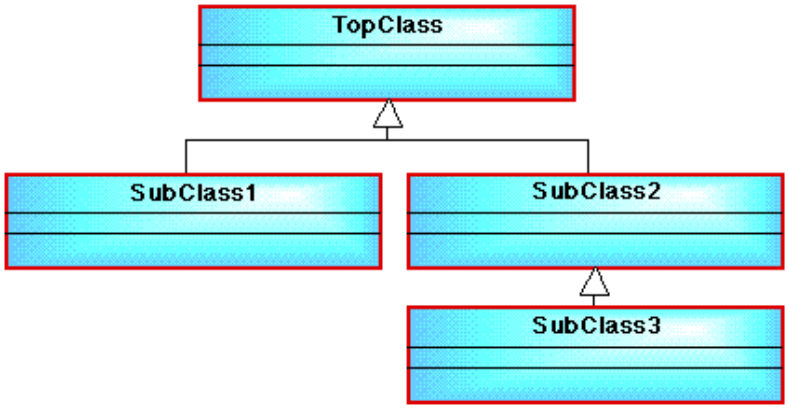

Figure 12: *Class hierarchy*

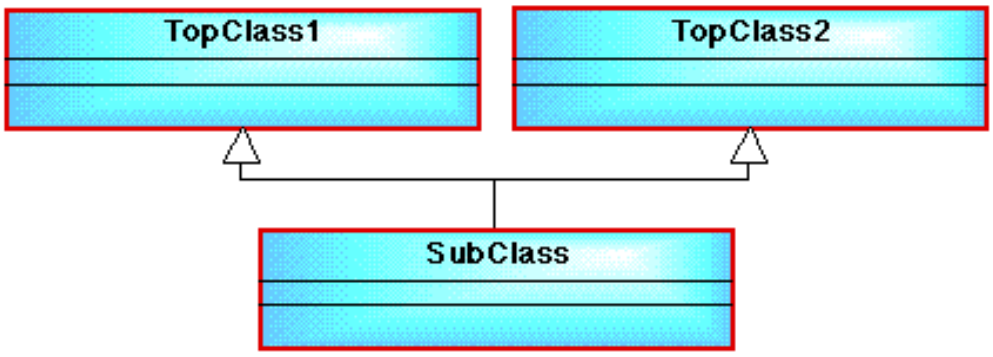

Figure 13: *Disallowed class hierarchy*

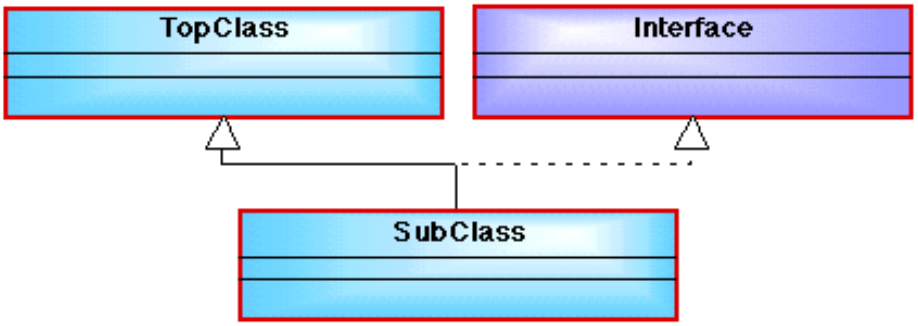

Figure 14: *Interface model*

VI. GUI Overview

A Graphical User Interface (GUI) is an interface between a user and a computer that makes use of devices such as windows, menus and buttons to facilitate input. GUIs are built of components. A component is an object with which the user interacts via mouse, keyboard or another form of input such as voice recognition.

The classes that create GUI components are part of the Swing GUI components package **javax.swing.** Most Swing components are written, manipulated and displayed completely in Java.

The original GUI components from the *Abstract Windowing Toolkit* package called **java.awt** are tied directly to the local platforms' graphical user interface capabilities. When a Java program with an AWT GUI executes on different platforms, the GUIs components display differently on each platform.

Swing components allow the programmer to specify a uniform look and feel across all platforms. In addition, Swing enables programs to provide a custom look and feel for each platform or even to change the look and feel while the program is running.

Swing components are often referred to as *lightweight components* because they are not weighed down by the complex GUI capabilities of the platform on which they are used. AWT components that are tied to the local platform's windowing system and rely on it to determine functionality are called *heavyweight components.* Several Swing components are still heavyweight components – in particular, subclasses of **java.awt.Window** that display windows on the screen and subclasses of **java.applet.Applet** still require direct interaction with the local windowing system.

Figure 15 shows an inheritance hierarchy of the classes that define attributes and behaviours that are common to most Swing components. Each class is displayed with its fully qualified package name and class name. Much of each GUI component's functionality is derived from these classes.

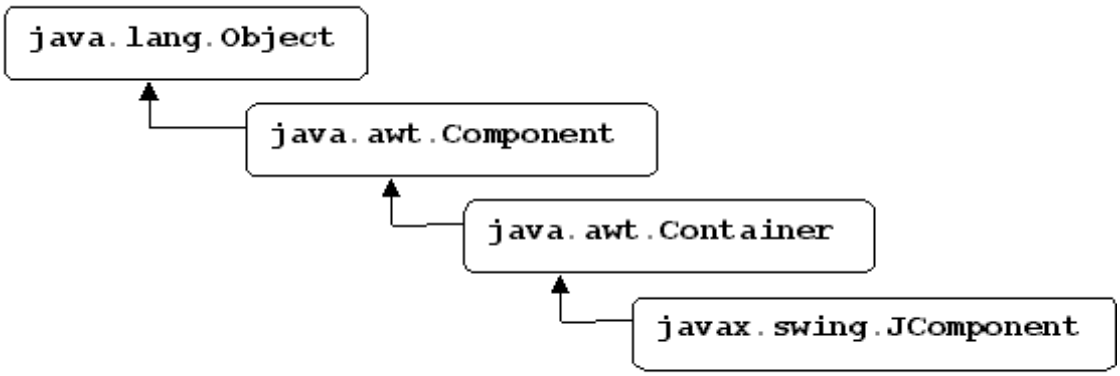

Figure 15: *Swing classes*

A container is a collection of related components. Class Container defines the common attributes and behaviours for all subclasses of Container. The class Container gives two methods:

1. add (Component comp) adds the specified component to a container (e.g. places a 'button' into a 'frame').

2. setLayout (LayoutManager mgr) sets the layout of components within a container according to its argument, e.g. in terms of its x-y cooridinates.

Remember that components cannot be displayed separately on the screen, but must first be placed in a container.

VII. Handling Events

GUIs are event driven (i.e. they generate events when the user of the program interacts with the GUI). Some common interactions are moving the mouse, clicking the mouse, clicking a button, typing in a text field, selecting an item from a menu, etc.

When a user interaction occurs, an event is sent to the program. GUI event information is stored in an object of a class that extends AWTEvent. There are three parts to the event handling mechanism – the *event source*, the *event object* and the *event listener*. The event source is the particular GUI component with which the user interacts. The event object encapsulates information about the event that occurred. This information includes a reference to the event source and any event-specific information that may be required by the event listener to handle the event. The event listener is an object that is notified by the event source when an event occurs. The event listener receives an event object when it is notified of the event, then uses the object to respond to the event. Different components have different listeners associated with them:

1. **Action Listener**: Buttons, text fields

2. **Item Listeners** : Check boxes, radio buttons

3. **Window Listeners**: Window containers

4. **Text Listeners** : Text areas

There are also other sorts of listener, for example, mouse listeners. The different kinds of listener are described by a set of interfaces which in turn implement the EventListener interface.

The programmer must perform two tasks to process a graphical user interface event in a program – register an *event listener* for the GUI component that is expected to generate the event, and implement an *event handling method* (or set of event handling methods). Commonly, event handling methods are called *event handlers.*
An event listener for a GUI event is an object of a class that implements one or more of the event listener interfaces from package **java.awt.event** and package **javax.swing.event**.

An event listener object listens for specific types of events generated by event sources (normally GUI components) in a program. An event handler is a method that is called in response to a particular type of event. Each event listener interface specifies one or more event handling methods that *must* be defined in the class that implements the event listener interface. Remember that interfaces define abstract methods. Any class that implements an interface must define all the methods of that interface; otherwise the class is an abstract class and cannot be used to create objects.

Required Reading

In **Chapter 6** of the text, read the following sections:

6.1-6.5, 6.8, 6.11 and 6.12

In **Chapter 11** of the text, read the following sections:

11.2-11.09, 11.11, 11.13

Useful web site links

1. Basic GUI tutorial, http://java.sun.com/docs/books/tutorial/uiswing/index.html (last visited 17 June 2007)
2. http://java.sun.com/docs/books/tutorial/uiswing/components/dialog.html#features, (last visited 17 June 2007)
3. http://www.apl.jhu.edu/~hall/java/Swing-Tutorial/Swing-Tutorial-JOptionPane.html, (last visited 17 June 2007)
4. http://gocsm.net/grant/CIS254/Resources/JOptionPane.html, (last visited 17 June 2007)
5. http://java.sun.com/docs/books/tutorial/uiswing/learn/index.html, (last visited 17 June 2007)
6. http://java.sun.com/docs/books/tutorial/uiswing/components/components.html, (last visited 17 June 2007)