# 12 Introduction to Objects

*This section is required for the final, optional, checkpoint-6. If you are going on to do Computer Simulation or have a general interest in computing, you are strongly advised to read this section now.*

## 12.1 The Basics

An *object* is a programming *building block* that is defined to have properties and a series of *methods* that modify these properties, returns information from the *object* or instruct the *object* to perform some task. This somewhat abstract concept is best looked at by taking one of the *objects* we have been working with, the `SimpleGraph` object. This object has

1. **Properties**: for example *colour* of the graph, *title* of the graph, list of points to be plotted etc.
2. **Methods**: to change these properties, for example `setColor()`, and to add data points and to instruct the object to perform tasks, for example `showGraph()`.

The internal workings of the object are hidden from the user who only creates instances of the object (using a *Constructor*), and manipulates it using its own *methods*. This allows the *object* to be a self contained and isolated from the programs that uses it. This has the advantage that the *object* can be used in many different programs, but more important is that the *object* can be tested, or replaced with an improved version (for example a more sophisticated one), and then provided that the new *object* has the same *methods* then the the main program will work with the new version.

This concept of having *methods* associated with the *object* and the internals of the *object hidden* and only accessible via *methods* is central to the "object oriented" programming approach. The problem of programming then becomes the design of *objects* to hold the data in a useful way complete with *methods* to manipulate and interact with the *objects*.

## 12.2 A simple `Point` object

Let us consider a simple example of a `Point` object which describes a three-dimensional point in space. A *point* can be described by three coordinates, typically its $x, y, z$ location, this defines how we wish to create a `Point`. Look at this first with the following partial definition.

```
public class Point {                                    // (1)

    private double xLoc, yLoc, zLoc;                    // (2)

    public Point(double x, double y, double z) {        // (3)
        xLoc = x;                                       // (4)
        yLoc = y;                                       // (5)
        zLoc = z;                                       // (6)
    }

    public Point() {                                    // (7)
```

```
        xLoc = 0.0;                                    // (8)
        yLoc = 0.0;                                    // (9)
        zLoc = 0.0;                                    // (10)
    }
<more to follow>
```

Consider the start of the definition of the `Point` object,

(1) Start of the definition of the *class* `Point`. Note the `public` keyword that means it can be seen externally.

(2) Declares three *internal* double variables (note the `private` keyword). These variables can only be see inside the class `Point`.

(3) Declares a *constructor* `Point` note the same name as the object. This takes three doubles, being the location of the `Point`.

(4-6) Sets the internal variables to the values supplied in the argument list.

(7) Is an alternative, or *overload* constructor that takes no arguments. This creates a `Point` at position $0,0,0$. Note you can have an many *overloaded* constructors as you like[1].

(8-10) Set the internal variables to zero.

So far this allows us to create a `Point` object, for example in our main program we can write,

```
        Point location = new Point(2.5 , 4.5 , 7.0);
        Point orgin = new Point();
```

which will create a `Point` object with coordinates $2.5,4.5,7.0$ called `location` and a second `Point` called `origin` at location $0,0,0$. However this does not allow us do anything.
We now need *methods* to work with the *object*. Let's first consider *methods* to return the $x,y,z$ location which are written as:

```
<definition>
    public double getX() {                             // (1)
          return xLoc;                                 // (2)
    }
    public double getY() {                             // (3)
          return yLoc;                                 // (4)
    }
    public double getZ() {                             // (5)
         return zLoc;                                  // (6)
    }
<more to follow>
```

which operate as:

(1) Define a method `getX()` which takes no arguments.

[1] Try adding a constructor to take two `doubles`, $x$ and $y$, and sets the third coordinate $z$ to zero.

(2) Return `xLoc`.

(3-6) Repeat for `yLoc` and `zloc`.

These *methods* allow, in our main program to write,

```
double xPosition = location.getX();
double yPosition = location.getY();
double zPosition = location.getZ();
```

where `location` is a `Point` as defined above. This allows us to read back the values of a `Point`, still not very useful.

**Note** the syntax. The `()` are essential since `.getX()` is a *method.*

*Aside: By convention methods that return the values is internal variables all start with* `get` *this is not actually required, but is considered good* JAVA *programming practice. Such methods are commonly referred to as "getters".*

Now lets as look as at a *method* to calculate the distance to the point from the origin which can be written as,

```
public double fromOrigin(){
    return Math.sqrt(xLoc*xLoc + yLoc*yLoc + zLoc*zLoc);
}
```

where since the *method* is defined within the class *Point* is has access to the `private` variables `xLoc`, `yLoc` and `zloc`.

This *method* allow us to write

```
double distanceFromOrigin = location.fromOrigin();
```

again note that the final `()` are essential.

Things become more useful when we are able to pass a parameter to the *method*, for example we wish to calculate the distance between two `Points`. This starts to introduce the real useful power of this technique. We want a *method* that will allow us to write

```
Point first = new Point(1.0,1.0,1,0);
Point second = new Point(2,0,2,0,2.0);
double distanceBetweenPoints = first.distance(second);
```

so we have to define a *method* that takes a second `Point` as a parameter. above. The code for such a *method* is:

```
public double distance(Point p) {                              // (1)
   double xDelta = this.xLoc - p.xLoc;                         // (2)
   double yDelta = this.yLoc - p.yLoc;                         // (3)
   double zDelta = this.zLoc - p.zLoc;                         // (4)

   return Math.sqrt(xDelta*xDelta+yDelta*yDelta +zDelta*zDelta); //(5)
  }
```

Again look at this line by line,

(1) Declares a *method* called `distance` that takes a `Point` object as an argument and returns a `double`.

(2) Declares a local double `xDelta` and sets it to the difference between the $x$ coordinates of the current `Point`, accessed by `this.xLoc`, and the `Point` passed as the parameter `p`, accessed by `p.xLoc`.

Note the use of the `this` keyword to ensures we are referring to the variables associated with the *current* point.

(3-4) Repeats (2) for the $y$ and $z$ coordinates.

(5) Return a double being the distance between the two `Point`s.

Now lets declare our final *method* to *add* two `Point`s together and return a new `Point`. This *method* has to take a second `Point` as a parameters and then return a `new Point`. The code for this is:

```
public Point add(Point p) {                          // (1)
    double xNew = this.xLoc + p.xLoc;                // (2)
    double yNew = this.yLoc + p.yLoc;                // (3)
    double zNew = this.zLoc + p.zLoc;                // (4)
    return new Point(xNew,yNew,zNew);                // (5)
}
```

Again line by line we have:

(1) Declare a *method* `add` that takes a `Point` as an argument and returns a `Point`.

(2-4) Declare three internal `double`s which are the location parameter `Point` (`p`), added to the current `Point`.

(5) Return a `new Point` which has the new coordinates just calculated.

This now allows us to write,

```
Point first = new Point(1.0,1.0,1,0);
Point second = new Point(2.0,1.0,-1.0);

Point third = first.add(second);
```

which is a much nicer, and more obvious, piece of code than having masses of $x, y, z$ locations in your main program.

Finally lets add a `toString` method to that return a formatted `String` containing the three coordinates, being,

```
public String toString() {
    return String.format("(%g,%g,%g)",
                         this.xLoc,this.yLoc,this.zLoc);
 }
```

now the `toString` method is a one of the standard methods defined for *all* objects and is automatically called when we try to convert an *Object* to a `String`. We have replaced (overloaded) it for our Point object, so in put main program we can now write

```
.....
Point third = first.add(scond);
System.out.println("Value of third point is : " + third);
```

When the `Point` is *added* to the string, it is automatically converted using a `toString()` using the method we have just written, and then concatenated, just as you would hope.

This simple example illustrated the basics of creating and using a new *object* and writing simple *methods* to manipulate and operate on it. This is only the start of *object oriented programming*, which will be expanded on in future courses, in particular the optional *Computational Simulation* next term, or *Computational Methods* next year.

## 12.3 Putting it together

Due to the filename constraints in JAVA we have to be careful in putting this together. Remember that the filename **must** match the *class* is contains, so this means that the `Point` class must be contained in a file called `Point.java`, while the class containing the `main` program **must** be in a different file.

In this case we have **two** files `Point.java` that contains,

```
public class Point {

    private double xLoc, yLoc, zLoc;

    public Point(double x, double y, double z) {
        xLoc = x;
        yLoc = y;
        zLoc = z;
    }
<definition of the methods>
}
```

and the `main` program that uses `Point`, for example,

```
   public class PointTest {
      public static void main(String args[]){

        Point first = new Point(2.0,4.0,6.0);
        Point second = new Point(-1.0,3.0,-5.0);

        System.out.println("Distance between first and second is :" +
                           first.distance(second));
        <---- rest of code ---->
    }
 }
```

which **must** be in a file called `PointTest.java`. When you compile `PointTest.java` using `javac` it will automatically look for a file called `Point.java` in your current directory which it will also compile to give a `Point.class` file. Then when you run `PointTest` with `java PointTest` is will automatically pick-up the code for the `Point` object, which is contained in the `Point.class` file.

This is the **most** basic method of using objects, you will learn much more about managing objects and packages in future courses.

## 12.4 Why bother with objects and classes

The first thing you notice when starting with objects is that you appear to write *more code* and it is *more complex* than if you did it all in the main program. Yes, to start with this is true, but the main program become massively simplified and if you define your objects carefully then they can be re-used and *extended* them for use in other programs. This saves you time and effort in the *long run*. More importantly the use of objects forces you to think in a structured way and start breaking the task down into manageable *chunks*. You can then define, write and more importantly test each object (and its associated methods) *before* you use them in your final main program[2]. Most programmers spend in excess of 90% of their time testing and debugging, so creating sensible *objects* in an object oriented programming language **is** actually a very efficient method of working.

There are articles, books and complete courses on structured programming, and *object oriented programming*, all of which are valid, however you only really *learn* and *understand* the power of this technique by trying it out, seeing what works, making mistakes and finally starting to think is the *correct way*; there is no substitute for sitting at the computer trying things out and making it work!

## Examples

Source code for the following on-line examples are available,

- Code for the `Point` object Point
- Code for the `PointTest` test program PointTest

### What Next?

You are now ready to try the final, optional checkpoint for these people who started at checkpoint 4 or for those people to are going on to the Computer Simulation course. In second part of this checkpoint you should try and modify the object `Point` discussed above as detailed in the checkpoint description.

[2]With large programs the different classes are ofter written by different programmers in a development team, this scheme allows each programmer to concentrate on their piece of the code allowing them to fit it all together at the end.