The 3-Dimensional Box Class and Its Viewer

In this exercise, you will define three classes that support 3-dimensional geometric operations:

**Point3D** encapsulates a 3-dimensional point with \( x, y, z \) coordinates.

**Size3D** encapsulates the measurement of dimensions parallel to the \( x, y, z \) axes.

**Box3D** describes a box region parallel to the \( x, y, z \) axes by encapsulating a **Point3D** object that represents the point with the minimum coordinates in the box and a **Size3D** object that represents the dimensions of the box.

These three classes illustrate many of the principles and techniques used to build classes. The **Point3D** and **Size3D** classes encapsulate primitive data (**double**) and illustrate the use of constructors, set functions, and get functions together with error checking when required. The **Box3D** class encapsulates object data (**location** of type **Point3D** and **size** of type **Size3D**) and has a much richer collection of member functions than the first two classes.

To test your work, we have provided the class **Box3DViewer** that opens a graphical user interface in which you may draw one or more boxes. The boxes are shown in perspective projection. This is accomplished by using the **SimpleProjector** class. Finally, the **PlotColors** class is a helper class that enables successive boxes to be drawn in different colors. A screen snapshot of the viewer is shown below.
1. The Point3D Class

The `Point3D` class will encapsulate member data `x`, `y`, `z` of type `double`. The class will be similar to the Java class `Point2D.Double` but we will not ask you to deal with inner classes, one for `Float` and one for `Double`, as in pure Java. We will simply ask you to design the `Point3D` class as a standard class.

The `Point3D` class should have the following constructors:

```java
public Point3D()
public Point3D(double x, double y, double z)
public Point3D(Point3D p)
```

The second and third constructors should call the corresponding `setLocation` functions given below rather than do the work directly.

The class should have the following member functions:

```java
public void setLocation(double x, double y, double z)
public void setLocation(Point3D p)
public void setX(double x)
public void setY(double y)
public void setZ(double z)
public double getX()
public double getY()
public double getZ()
```

Be careful to use the qualifier `this` to distinguish the parameter names that are exactly the same as the corresponding member data names. Also, in `setLocation(Point3D p)`, you must test to make sure that `p != null`.

**Additional comments:** The abstract Java class `Point2D` from which the class `Point2D.Double` is derived requires the definition of a pair of functions:

```java
public boolean equals(Object object)
public int hashCode()
```

It would be relatively easy for you to write the `equals` function here following the model used in the `SimpleTime` exercise. We do not however want to get into the question of how to write the corresponding `hashCode` function using the `double` data in the member variables `x`, `y`, `z`. Therefore we are skipping these functions. We are also skipping the various `distance`
functions in the `Point2D` class that are straightforward to define but would add more tedium to this class definition.

2. The Size3D Class

The `Size3D` class will encapsulate member data `xSize`, `ySize`, `zSize` for storing dimensions. As a requirement of this class, your member functions should always ensure that these values are positive, that is, greater than or equal to zero. The reason for this requirement is that negative dimensions do not make sense.

The `Size3D` class should have the following constructors:

```java
public Size3D()
public Size3D(double xSize, double ySize, double zSize)
public Size3D(Size3D size)
```

The second and third constructors should call the corresponding `setSize` functions given below rather than do the work directly.

The class should have the following member functions:

```java
public void setSize(double xSize, double ySize, double zSize)
public void setSize(Size3D size)
public void setXSize(double xSize)
public void setYSize(double ySize)
public void setZSize(double zSize)
public double getXSize()
public double getYSize()
public double getZSize()
public double getVolume()
```

Be careful to use the qualifier `this` to distinguish the parameter names that are exactly the same as the corresponding member data names. Also, in `setSize(Size3D size)`, you must test to make sure that `size != null`.
We envision the following relationships between the constructors and the member functions:

```java
public Size3D(double xSize, double ySize, double zSize)
    calls
public void setSize(double xSize, double ySize, double zSize)
```

and

```java
public Size3D(Size3D size)
    calls
public void setSize(Size3D size)
    which calls
public void setSize(double xSize, double ySize, double zSize)
```

In turn:

```java
public void setSize(double xSize, double ySize, double zSize)
    calls
setXSize(double xSize), setYSize(double ySize), setZSize(double zSize)
```

This means that the error checking (making sure that the values assigned to the member data are positive) can be concentrated in the last three functions. In each of these functions, you should do an `if`-test or use the `?:` syntax to guarantee that the parameter is assigned only if it is greater than or equal to zero. If the parameter is negative then zero should be assigned.

The `Volume` function should return the product of the three sizes.

### 3. The Box3D Class

The `Box3D` will represent a box parallel to the coordinate axes by using two member data variables:

- `Point3D location`
- `Size3D size`

Each of these member data variables should be initialized in the member data declaration area using the `default constructor` for its class. During the operation of the `Box3D` class, the values of these member data variables may be changed by calling member functions of their class but the variables themselves must not be changed by using assignment.
The `location` variable will represent the corner of the box whose coordinates have the minimum values. The `size` variable will represent the dimensions of the box.

The `Box3D` class should have the following constructors:

```java
public Box3D()
public Box3D(Point3D location, Size3D size)
public Box3D(Box3D box)
```

The second and third constructors should call the corresponding `setBox` functions given below rather than do the work directly.

The `Box3D` class should have the following set and get functions;

```java
public void setBox(Point3D location, Size3D size)
public void setBox(Box3D box)
public void setLocation(Point3D location)
public void setSize(Size3D size)
public Point3D getLocation()
public Size3D getSize()
```

The first `setBox` function should simply call `setLocation` and `setSize` by passing the parameters `location` and `size` respectively. The second `setBox` function should just return if the parameter `box` is `null`. If `box` is non-null, then it should pass `box.location` and `box.size` to the first `setBox` function. In this way, the calls are channeled through the same pathway. This reduces the possibility of similar but independent bugs.

Each of the functions `setLocation` and `setSize` should do nothing if passed a `null` parameter. If the parameter is non-null then the data in the parameter must be copied into the corresponding member data variable. *Under no circumstances, should you use an assignment statement.* Using an assignment will make an internal member data variable reference an entity that is outside of the control of the object and will permit code outside of the object to change this member variable. This introduces enormous and unacceptable potential for bugs.

We will illustrate what you should do using `setLocation` as an example:

```java
public void setLocation(Point3D location) {
    this.location.setLocation(location);
}
```
What exactly is the *magical* line of code in the body of `setLocation` doing? First, since the parameter name `location` is the same as the name of the member data `location`, we must do something to tell them apart. By writing `this.location`, we specify the member data variable. Since the `Point3D` class already has a `setLocation` function that will copy the data from another `Point3D` object, we can invoke this function on the member data variable:

```
this.location.setLocation( ... );
```

What parameter actually gets passed into this call of `setLocation`? None other than the `location` parameter that was passed into the `setLocation` function of the `Box3D` class!

What is happening is that the member function `setLocation` of the `Box3D` class is delegating its work to the member function of the same name of the `Point3D` class as activated on the member data variable specified as `this.location`. Naturally, this process is called `delegation`. Delegation is an extremely common technique in object-oriented design.

You may have noticed that the `Box3D setLocation` function does *not* check if `location` is `null`. This is because the check is performed in the `Point3D setLocation` function.

We will now describe the additional member functions of the `Box3D` class that perform more significant operations than the simple set and get functions. The first group of these functions determines if a `Point3D` object is inside a `Box3D` object and if desired will expand the `Box3D` object to include a specified `Point3D` object or an array of `Point3D` objects.

```
public boolean isInside(Point3D p)
public void expand(Point3D p)
public void expand(Point3D[] data)
public void expand(Point3D[][] data)
```

First of all, as always, all functions must check for `null` parameters. In the case of `isInside`, the question arises as to what to return if the parameter `p` is `null`. Since a `null` point does not exist, the sensible choice is to return `false`.

The `isInside` function and the first `expand` function must do significant work. These functions are not one-liners. In both functions, it is necessary to *extract the coordinate information* from the member data variables `location` and `size` and from the parameter `p`. Then, in the `isInside` function, one must determine by testing appropriate inequalities involving the coordinates whether or not the point `p` is in the box. In the `expand` function, one must determine if the `location` and `size` must be adjusted to create a larger box to include the point `p`. 
**More Detailed Hint:** Let's just consider x-coordinates. Then you will need to extract:

```java
    double xmin = location.getX();
    double xmax = xmin + size.getXSize();
    double x = p.getX();
```

Then, among the tests you will make in `isInside` are:

```java
    (xmin <= x) && (x <= xmax)
```

You will need, in fact, to chain together 6 inequalities using `&&` in order to decide if the point `p` is in the box. In `expand`, you may need to modify `xmin` downward or `xmax` upward. In any case, at the end, you will need to call `location.setLocation` and `size.setSize` to save the modifications (being quite careful as to what you pass to these functions).

In the function, `void expand(Point3D[] data)`, you need to run a loop and call the single point `expand` function on each point in the data array. Thus, the 1-dimensional array version of `expand` calls the single point version of `expand`. Similarly, in `void expand(Point3D[][] data)`, you must again run a loop but this time the call will invoke the 1-dimensional array version of `expand`. Thus, in the `expand` family of functions, each function invokes the next simpler function in the family until at the level of a single point the actual work must be done. This is an instance of the principle of recursion in which a complex task is performed by doing the same task in a simpler situation until at the simplest level of the problem the work is done in detail for the easiest case.

In the `SimpleProjector` class that is supplied with this exercise, you may examine two families of functions, `transform` and `perpTransform`, which use the principle of recursion to extend an operation on a single point to 1-dimensional and 2-dimensional arrays of points. These examples will illustrate the Java details you need to know to complete the definitions of the `expand` functions in `Box3D`.

The final member functions deal with constructing arrays that contain the 8 vertices (corners) of the box or the 12 edges of the box. It is necessary to have a helper function:

```java
    /**
     * Return the Point3D constructed from the given coordinates
     * (u, v, w) using the calculation:
     * <CODE><UL>
     *   <LI>x = xLocation + u * xSize
     *   <LI>y = yLocation + v * ySize
     *   <LI>z = zLocation + w * zSize
     * </UL></CODE>
    */

    public Point3D PointInBox(double u, double v, double w)
```
The purpose of this helper function is to describe points in the box using a standard set of “coordinates” u, v, w where these “coordinates” range from 0 to 1. In particular, you will then be able to list the 8 vertices of the box as:

```
PointInBox(0, 0, 0)
PointInBox(1, 0, 0)
PointInBox(1, 1, 0)
PointInBox(0, 1, 0)
PointInBox(0, 0, 1)
PointInBox(1, 0, 1)
PointInBox(1, 1, 1)
PointInBox(0, 1, 1)
```

Here the first four points represent the bottom of the box and the last four points represent the top of the box.

The notation in the comment for `PointInBox` is descriptive but is not the complete Java code. The Java code for the `x` part of the computation would in fact read:

```
double xLocation = location.getX();
double xSize = size.getXSize();
double x = xLocation + u * xSize;
```

Once the new coordinates x, y, z have been computed, one must create a new `Point3D` object and so of course the `new` operator will have to be used. Thus, the final line of the function `PointInBox` will be:

```
return new Point3D(x, y, z);
```

Finally, we come to the last two member functions, the ones that return a list of the 8 vertices or a list of the 12 edges:

```
public Point3D[] getVertexList()
public Point3D[][] getEdgeList()
```

Notice that `getEdgeList` returns a 2-dimensional array since each edge is 1-dimensional array with two endpoints and we must collect these 12 edge arrays into the full edge list.

To illustrate the Java syntax, we will give you the code for `getVertexList()`:
public Point3D[] getVertexList() {
    return new Point3D[] {
        PointInBox(0, 0, 0),
        PointInBox(1, 0, 0),
        PointInBox(1, 0, 1),
        PointInBox(0, 1, 0),
        PointInBox(0, 0, 1),
        PointInBox(1, 0, 1),
        PointInBox(1, 1, 0),
        PointInBox(1, 1, 1),
        PointInBox(0, 1, 1)
    };
}

This construction is similar to many code sections you have seen in the GUI portions of your programs and you will see similar code (with different data types of course) if you examine the main program Box3DViewer.

We will leave the formulation of getEdgeList to you.

4. Concluding Remarks

When you download the Box3D project, you will not find the files Point3D.java, Size3D.java, and Box3D.java. You will have to create these files and add them to your project. Then you will need to program all three classes from scratch using the method naming conventions given above. If you change the function names or parameters, then your code will not compile with the test files SimpleProjector.java, PlotColors.java, and Box3DViewer.java. This will be a disaster. In general, on software projects in industry, once the method names and parameters of a class have been decided no changes are permitted unless everything on the project is informed and agrees to the changes. You should consider the names and parameters for this Box3D project frozen.

If you want to add challenge to this exercise, then go into the file SimpleProjector.java and delete all the code in the bodies of the methods. Then figure out the algebraic and geometric issues involved to compute a perspective or perpendicular projection and recreate the required code on your own. If you are ambitious but not quite so ambitious then at least read the code of SimpleProjector.java.

For more information on graphical transformations, take a course on computer graphics in the future!