COMP 401

Prasun Dewan

Composite objects and shapes

In the previous chapter, we gained some experience with declaring classes and interfaces, but relatively little experience with using objects. We essentially wrote a main that tested the getters and setters, making sure that the dependent properties were calculated correctly. In this chapter, we will gain more experience with both defining and using objects by creating objects that are "composed" of smaller ones. Thus, we will gain experience with declaring the classes and interfaces of the composed objects and creating and manipulating the smaller, component objects. The composed objects will themselves become components of larger objects, much as letters are composed into words, words into sentences, sentences into paragraphs, and so on. As in the previous chapter, the objects will mainly be graphics shapes, though now we will see that a composed object can contain both graphics and text properties. This will lead us to the distinction between structured and atomic types and physical vs. logical structure. We will see the importance of initializing object variables in constructors.

Location as a Point object

The line and other shapes we defined in the previous chapter represented their location as int X and Y coordinates, as shown in the reproduced line interface below:

```
public interface Line {
        public int getX();
        public void setX(int newX);
        public int getY();
        public void setY(int newY);
        public int getWidth();
        public void setWidth(int newVal);
        public int getHeight();
        public void setHeight(int newHeight);
}
```

An alternative interface is to represent the location as an instance of the Point interface we defined earlier:

```
public interface LineWithObjectProperty {
        public Point getLocation();
        public void setLocation(Point newLocation);
        public int getWidth();
        public void setWidth(int newVal);
        public int getHeight();
        public void setHeight(int newHeight);
}
```

Thus, this interface replaces the two properties, X and Y, of type **int**, with a single property, location, of type Point. So far, we have created object types whose properties were all primitives. This is the first object type that has a property that is not a primitive, that is, is an object.
The following code shows how this object property is implemented.

```
public class ALineWithObjectProperty implements LineWithObjectProperty {
        int width, height;
        Point location;
        public ALineWithObjectProperty (Point initLocation, int initWidth, int initHeight) {
                location = initLocation;
                width = initWidth;
                height = initHeight;
        }
        public ALineWithObjectProperty()  { }
        public Point getLocation() {return location;}
        public void setLocation(Point newVal) {location = newVal;}
        public int getWidth() {return width;}
        public void setWidth(int newVal) {width = newVal;}
        public int getHeight() {return height;}
        public void setHeight(int newHeight) {height = newHeight;}
}
```

The constructor now takes a single Point value rather than two int coordinates to describe the location. (Ask students to implement a constructor taking two int coordinates.). The constructor value is stored in a Point instance variable. By using an interface to type our constructor parameter and instance variable, we allow both instances of ACartesianPoint and APolarPoint to be used for the location. The remaining code is straightforward and follows the pattern we have seen before for independent properties – the getter of the property returns the value of the associated instance variable and the setter changes the variable. The following code illustrates how we might instantiate this clsas and display it using bjectEditor:

```
  lineWitobjectProperty lineWithobjectproperty =
        new ALineWithObjectProperty(new ACartesianPoint (10, 10), 20, 20);
   bjectEditor.editlineWithbjectPrpery);
```

Structure as a Type Classification Dimension

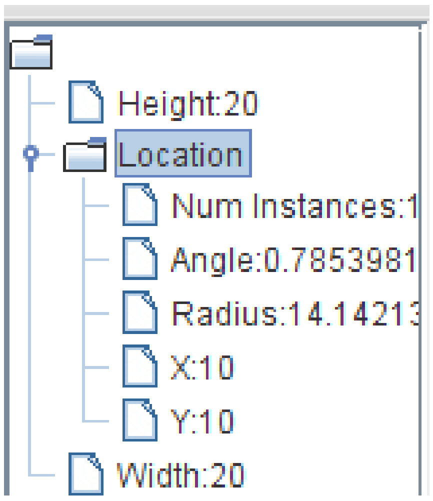
This example leads us to create a new way to classify types. So far, we have classified using two dimensions: predefined vs. programmer-defined, and primitive vs. object. Recall that predefined types are those that come with a Java system; others are programmer-defined types. Some of the examples of predefined types we have seen are int, double, Integer, Double, String, and Scanner. Examples of programmer-defined types: BMISpreadsheet, ABMISpreadsheet, AnotherBMISpreadsheet, Point, ACartesianPoint, and APolarPoint. Object types are classes and interfaces, the rest are primitive types.

Primitive type examples: int, double, boolean. Object type examples: Point, ACartesianPoint, String, Integer, Double. In Java, all programmer-defined types are object types, but some predefined types such as String are object types.

This chapter requires a new dimension for classifying types, based on the structure of their instances. Types whose instances can be decomposed into one or more component values are *structured types*; otherwise they are *atomic types*. int, Integer, double and Double are atomic types while Point and ACartesianPoint, are structured types. All primitive types such as int and double are atomic, as are some object types such as Double and Integer. This new dimension raises the following question: What criteria do we use to decompose program-defined values? Why are Integer and Double atomic and Point and ACartesianPoint structured?

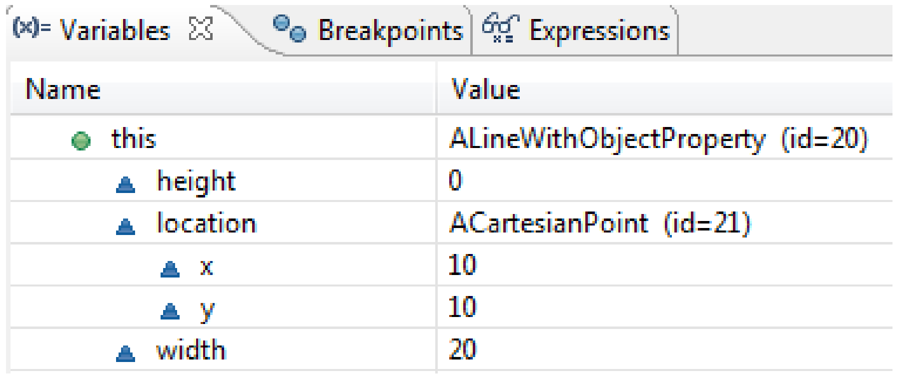
Logical vs. Physical Structure

One way to decompose an object is by its logical components, derived from the headers of its public methods. We can decompose an object into the properties extracted from these methods. If any of the properties is assigned an object, then we can recursively follow the same procedure to decompose it. This is the decomposition performed by ObjectEditor, as shown by the treeview it creates for the instance passed to it in the edit call above. In this example, the object property itself has properties, leading to a hierarchical structure in the tree view.



The object is decomposed into its three properties, Location, Height, and Width. Height and width are primitive, and hence cannot be further decomposed. Location is assigned an instance of AcartesianPoint – so we can further decompose into four atomic properties of type double. The logical structure of an object is what is exported by to its users as its external structure, through its public methods.

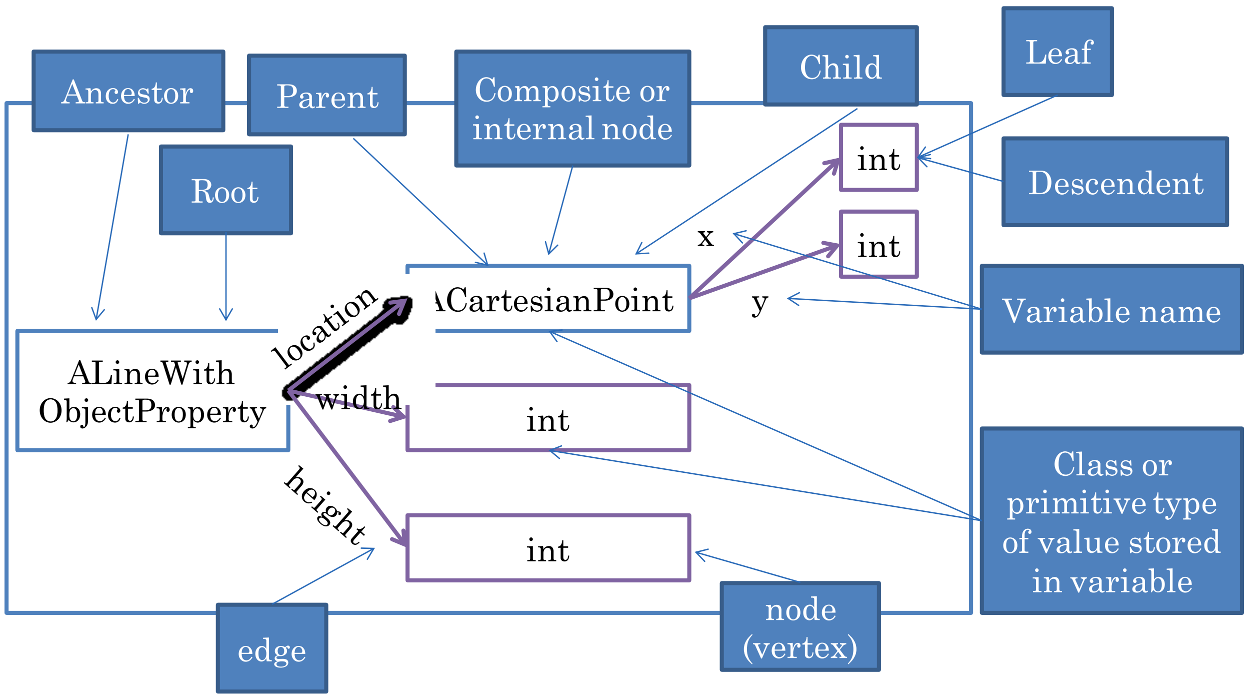
There is a second way to decompose an object, in which we decompose an object into its instance variables. If any of these variables is assigned an object, then we recursively follow the same procedure to decompose it. This is the decomposition perfomed by a debugger, as shown by the treeview it creates for the instance passed to bjectEditor in the edit call above. The physical structure of the object is essentially the structure of its representation in memory. It depends on the implementation of the object and its sub-objects.

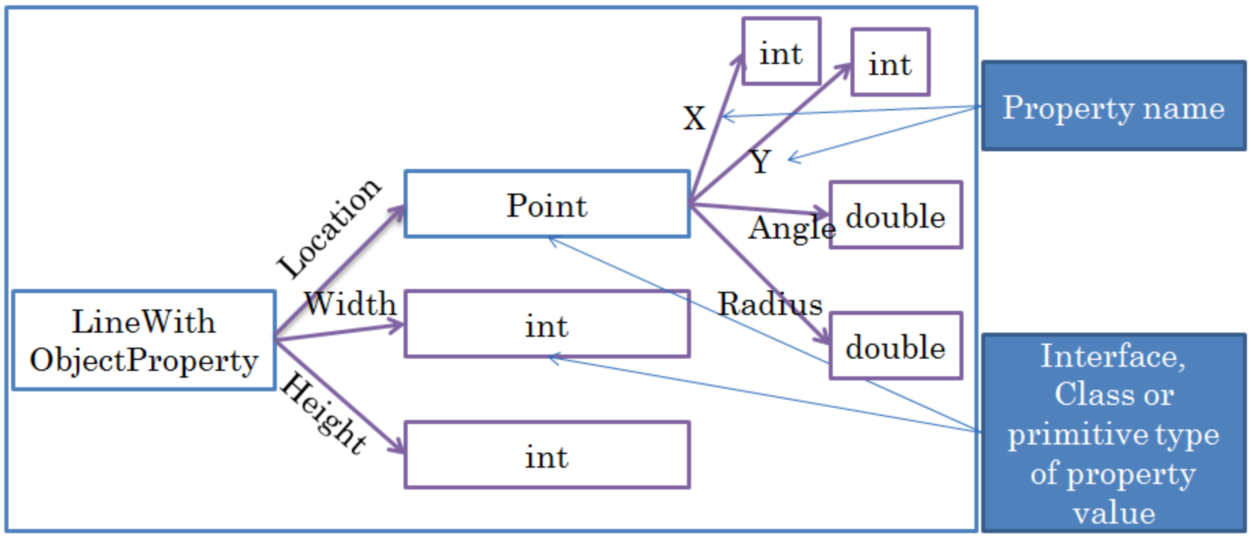


Name	Value
this	ALineWithObjectProperty (id=20)
height	0
location	ACartesianPoint (id=21)
x	10
y	10
width	20

Again, the object is decomposed into height, weight, and location, as these are not only properties but also instance variables. Again, height and weight are atomic. The main difference is that the object assigned to location is decomposed into two components, x and y, instead of four. This is because ACartesianPoijnt has two instance variables, storing Cartesian coordinates, instead of four properties, storing both Cartesian and Polar cordinates.

The views created by objectEditor and the eclipse debugger are difficult to draw by hand, so we will develop our own mechanisms to draw the two two structures. Figure ??? and ??? capture the physical and logical structure, respectively, of the instance of AlineWithobjectproperty we created above.





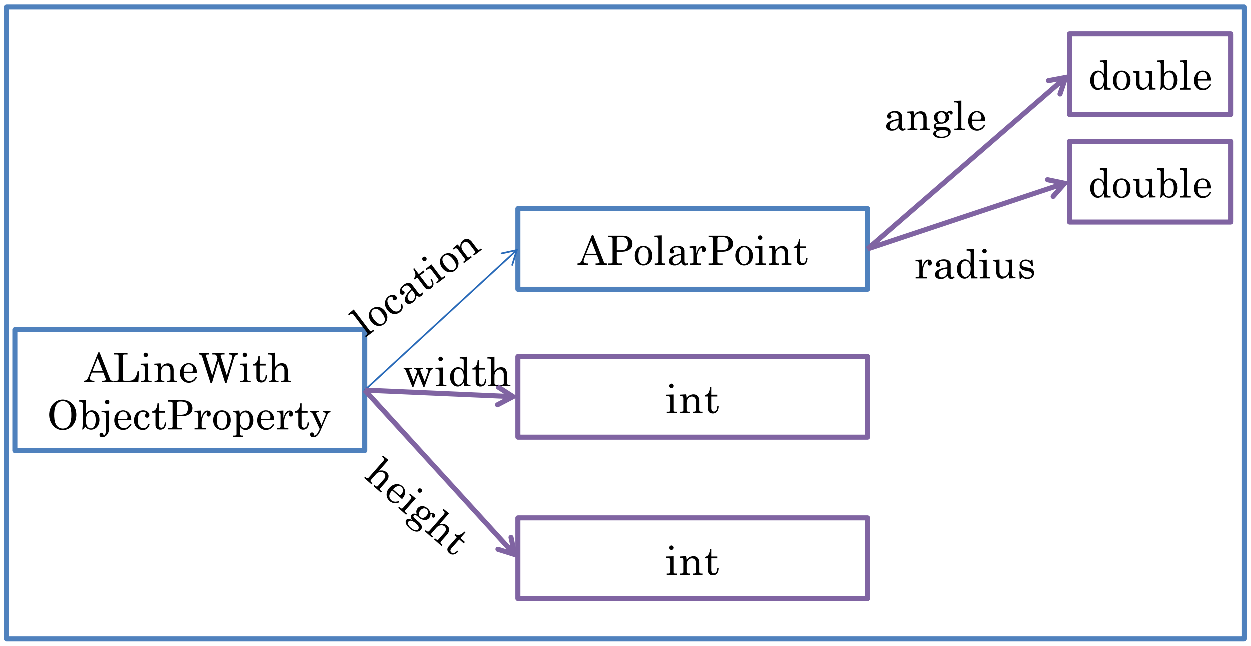
In both approaches, we create labelled diagrams connecting boxes, called nodes, using directed lines called *edges* or *links*. These diagrams resemble family tree diagrams, and borrow some of the terminology from the latter. For each object to be decomposed, we create such a diagram. In such a diagram, a node with no incoming edge is the *root*, which represents the object to be decomposed. An edge is drawn from some node a to a node b, if b is a component in the structure of a. Node a is called a *parent* of b and b a *child* of a. Nodes with no outgoing edge are *leafs*, which represent non decomposable atomic objects. Non leaf or root nodes are *internal nodes*. If a node *b* can be reached through a series of directed edges from a, then a is an ancestor of b, and b a *descendent* of a. Each node has a label describing its type. Each edge from a parent to a child has a label naming the child.

In the case of physical structures, we use classes and primitives as node types or labels, and instance variables as edge names. An edge e is drawn from a node a to b, if an instance variable e of type is declared in class a. For example, an edge labelled location is drawn from class Anewithbjectproperty to a node labelled AcartesianPoint.

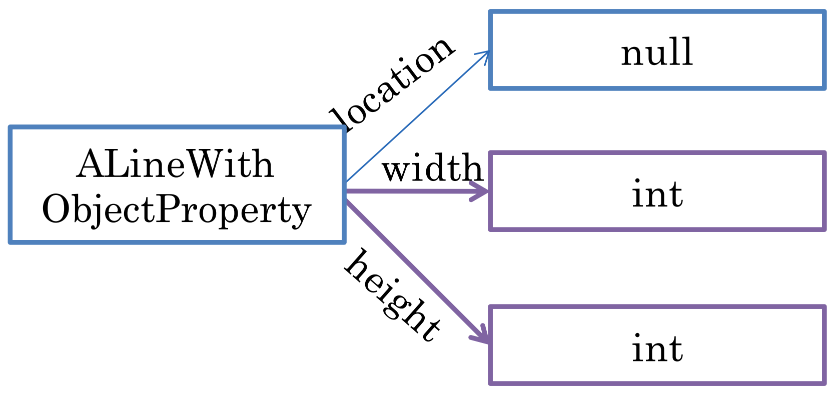
In the case of logical structures, we use interfaces and primitives as node types or labels, and properties as edge names. Thus, an edge e is drawn from a node a to b, if property e of type is b defined by interface a. For example, an edge labelled location is drawn from interface linewithbjectproperty to a node labelled Point. This approach assumes that each class has a unique interface that has all of its public instance methods. If this is not the case for some class c, then we use its name as the label of the corresponding node.

Instance-Specific Structure

Can two objects of the same class have different physical or logical structure? The answer is easy with physical structures. Polymorphism allows an instance variable in some class C to be assigned instances of different classes, leading to different physical structures of different instances of C. This is illustrated by comparing the physical structures shown in Figure ?? and Figure ??? of instances of the same class. In one case AcartesianPoint was assigned to location and in another case APolarPoint. As a result, in one case we have two int leaf nodes, and in the other, two double nodes.



It may seem that logical structures of two different instances of the same class would be the same. However, this is not the case. When we study new kinds of polymorphism introduced by inheritance, we will see that it is possible for objects with two different interfaces to be assigned to the same instance variable and property. A simpler reason for instance-specific logical and physical structures is that any object instance variable and property can be assigned the null value or an object, leading to different decompositions in the two cases. Figure ??? shows the logical structure of an instance of a Linewitbjectproperty in which the location has been assigned null.



Structured and Composite Types

As users of a type, we are interested in its logical structure, and as its implementers, we are interested in its physical structure. We will use the term object component to refer to both a physical instance variable and a logical property and use context to resolve its meaning.

We will refer to object types with one or more properties as *logically structured*, and those with one more instance variables as *physically structured*. A type that is not logically (physically) structured is logically (physically) atomic. Thus, the types AcartesianPoint is both logically and physically structured, as it has both instance variables and properties. The type Integer is physically structured as it has an instance variable holding an int value. However, it is logically atomic, as it has no properties, and essentially represent the encapsulated value. (Have students define my Integer and then MyNumber);

Again, we will use the term structured type to refer to either a physically structured or logically structured type, using context to disambiguate. It is rare for a type to be structured logically but not physically – wrapper types are perhaps the only important exceptions.

A type that has at least one component whose logical/physical component is itself logicallly/physically structured is a logically/physically composite type. Thus, ALine is not logically/physically composite since all of its components are primitive types. On the other hand, AlineWithObjectProperty is both logically and physically composite since its Location property and location variable are logically and physically structured, respectively.

As we will see later, the more the number of internal nodes in the logical structure of an object, the more reusable the code defining it.

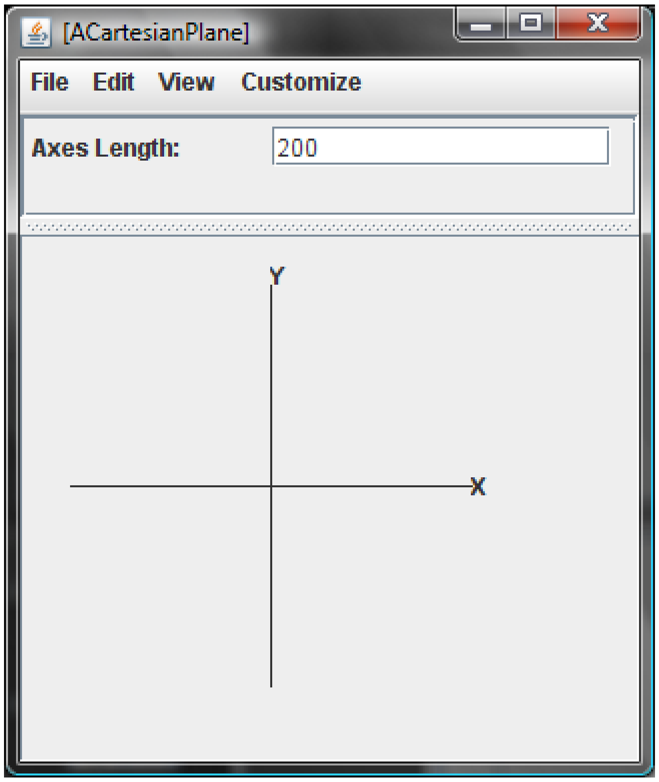
ObjectEditor Location-based Rules

While AlineWithObjectProperty and Aline have different interfaces and are different kinds of types, they essentially define two different logical representation of the same geometric object – a line. Thus, ObjectEditor displays instances of both of them as lines, though, of course, the tree views of the two kinds of objects are different, as shown in Figures ??? and ???. ObjectEditor allows the location of each bounded shape we saw before (Line, Rectangle, String, Image, Oval) to be stored in a point Location property type, or a pair of int X and Y properties. The type of the Location property must follow the conventions for a point we saw before – it must have getters for the int X and Y properties.

Choosing between the two alternatives for the location involves a tradeoff. If an object has a point Location property, then the code to move it becomes more reusable. The reason is that the location data can be passed as a single parameter to a method and more important, can be returned by a method. This is not the case if it has int X and Y properties.   On the other hand, a Location-based shape is less efficient to move. The reason is that ObjectEditor assumes that a point is immutable. As a result, moving the shape involved assigning a new object to the Location property, and processing of this object by ObjectEditor. This overhead is not incurred with an XY-based shape. Moreover, ObjectEditor has been tested more thoroughly with XY-based shapes. Thus, it is recommended that you continue to create such shapes.

Composite Object vs. Shape

While an instance of AlineWithObjectProperty is a composite object, the abstract geometric shape it represents is atomic in that it cannot be decomposed into smaller shapes. (One could decompose it into the points on the line, but that would require an infinite number of points for an (abstract rather than computer-based) line). What if we wanted to display a composite shape, that is, shapes whose components are smaller shapes? Given lines and other atomic shapes, it is possible to generate more complex geometries. This is illustrated in the graphics windows of Figure ???, which shows a Cartesian Plane, consisting of labelled X and Y axes of the same length. The main window displays an integer that determines the length of the X and Y axes. Changing this value automatically resizes the two axes, while maintaining the origin of the plane.



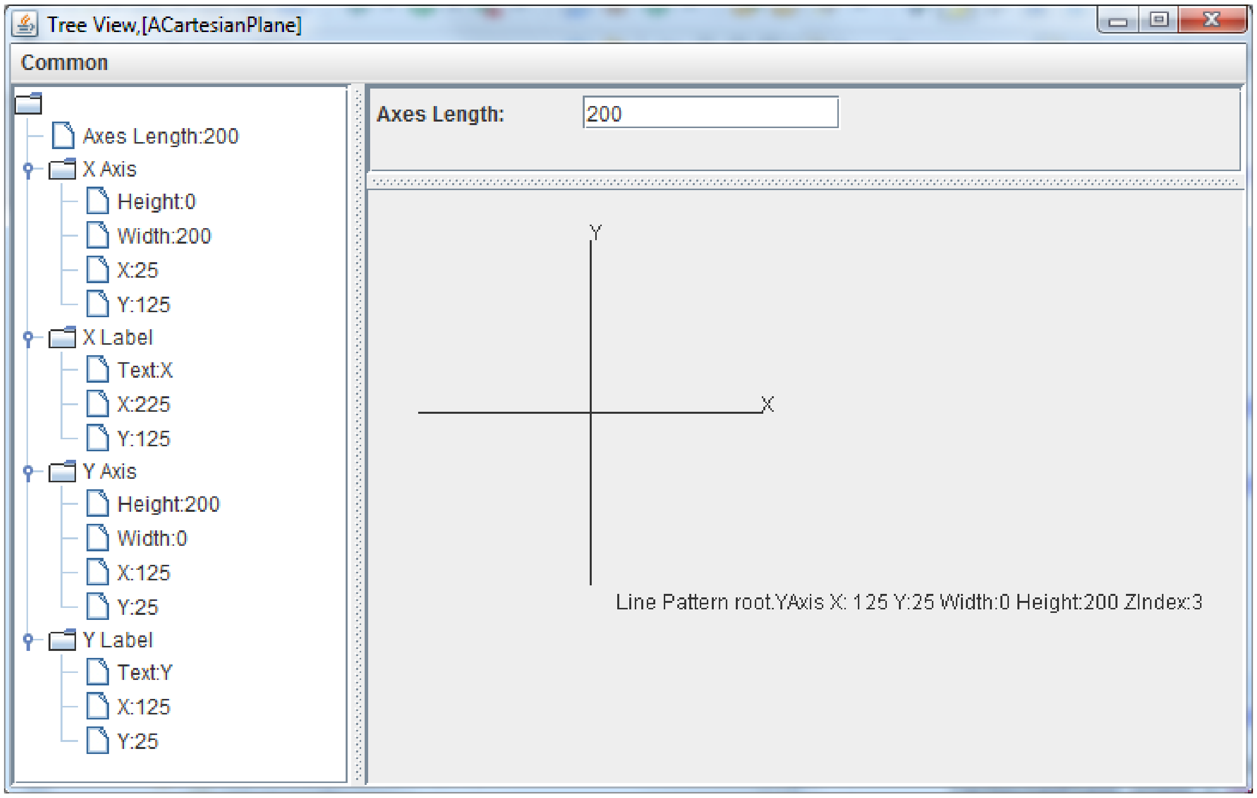
Axes Length: 200

How can we model the object displayed and manipulated in the user-interface? As we have learnt, there are many possible logical and pysical representations/structures for any real-world object. One logical representation would consist of the following properties:

- Editable, Independent AxesLength (int)
- Readonly, Dependent XAxis (Line)
- Readonly, Dependent YAxis (Line)
- Readonly, Dependent XLabel (StringShape)
- Readonly, Dependent YLabel (StringShape)

The editable AxesLength property holds an integer that can be edited to magnify or shrink the plane. The Xaxis and Yaxis properties are line shapes, representing the two perpendicular lines, and Xlabel and Ylabel are string shapes labelling the two lines. The AxesLength property, of course, is editable, but the other properties are not - they are readonly properties dependent on AxesLength. Of course we can model each of the component shapes in many ways – as before, we will simply use the representations that ObjectEditor understands. In fact, we will reuse types we created earlier – Line, ALine, StringShape, AstringShape.

Figure ??? illustrates this logical decomposition.



Axes Length:200
X Axis
Height:0
Width:200
X:25
Y:125
X Label
Text:X
X:225
Y:125
Y Axis
Height:200
Width:0
X:125
Y:25
Y Label
Text:Y
X:125
Y:25

Axes Length: 200

Line Pattern root.YAxis X: 125 Y:25 Width:0 Height:200 ZIndex:3

Based on the decomposition above, we can define the interface of our object (add it) based on the previously defined Line and StringShape interfaces.

The implementation of the interface is far more difficult than other classes we have seen so far. We will allocate an instance variable for each property:

```
 int axesLength;
 Line xAxis;
 Line yAxis;
 StringShape xLabel;
 StringShape yLabel;
```

Thus, we will create instance variables for not only the independent property – axesLength – but also the four dependent properties. In other words, all of our properties will be stored, as in the case of AnotherBMISpredsheet. We will see below the reason for our choice.

Final Variables Initialized at Runtime

In addition, we will create unexported state in the form of the origin X and Y of the plane.

```
final int originX, originY;
```

This variable is like a named constant in that it's value does not change once it has been initialized. However, a named constant is given a value when it is declared, at program writing time. This variable, on the other hand, is given a value in a constructor at runtime. In Java, it is possibly to create uniitialized final variables, but these must be initialized in constructors.

It is also possible to create final local variables in a method, which are initialized once. Most local variables tend to be assigned once – so it is a good idea to make such variables also final. However, it does increase the overhead of declaring them and the length of a code line, separating Java even more from Python, where a variable is not even declared. As space is a premium in both slides and this document, we will omit the final keyword. However, in production code, it is highly recommended you use the final keyword when appropriate.

Constructing, Initializing and Setting Dependent Objects

Calculating the value of a dependent primitive property was fairly straightforward. We simply wrote a dependency function to return a single value. With a dependent object property, we may have to write multiple dependency functions, one for each dependent component of the object. This is illustrated in the constructor below, which initializes four object properties, xAxis, yAxis, xLabel, and yLabel. For each of these properties, it calls two different functions to assign the x and y coordinates of the property based on the axes length. For example, for the xAxis, it calls the functions, toXAxisX() and toXAisY(), which compute the X and Y coordinates, respectively, of the X axis, based on the axes length.

```
public ACartesianPlane (int theAxesLength,
                        int theOriginX, int theOriginY ) {
   axesLength = theAxesLength;
   originX = theOriginX;
   originY = theOriginY;
   xAxis = new ALine(toXAxisX(), toXAxisY(), axesLength, 0);
   yAxis = new ALine(toYAxisX(), toYAxisY(), 0, axesLength);
   xLabel = new AStringShape ("X", toXLabelX(), toXLabelY());
   yLabel = new AStringShape ("Y", toYLabelX(), toYLabelY());
 }
```

In the case of a primitive stored property, we simply assigned the computed property value to the associated instance variable. The same assignment was made in both the constructor and the setter of each independent property. Thus, in the case of AnotherBMISpreadsheet the following assignment:

```
bmi = calculateBMI(height, weight);
```

was made in the constructor of AnotherBMISpreadsheet and the setters of Height and Weight

In the case of a stored dependent object property, in the constructor, we must first create an object, and then assign appopriate values to its properties, as we see below:

```
xAxis = new ALine(toXAxisX(), toXAxisY(), axesLength, 0);
```

In the setter for the independent property, we do not create new objects. Instead, we change the properties of the objects created in the constructors.

```
public void setAxesLength(int anAxesLength) {
   axesLength = anAxesLength;
   xAxis.setWidth(axesLength);
   yAxis.setHeight(axesLength);
   xAxis.setX(toXAxisX());
   xAxis.setY(toXAxisY());
   ….
}
```

Thus, the setter of our Cartesian plane object calls setters of properties of the object – which is typical for composite objects.

Code Duplication vs. Temporary Inconsistency

As we see above, the calls to toXAxisX() and other dependency computing functions) are duplicated in both the constructor and the setter of the independent property. This is good news in that we reused the functions in the constructor and setters. However, we can write even cleaner code by simply constructing the objects in the constructors, with inconsistent components, and then calling setters to make these components consistent. This approach is shown below:

```
public ACartesianPlane (int theAxesLength,
                        int theOriginX, int theOriginY ) {
   originX = theOriginX;
   originY = theOriginY;
   xAxis = new ALine();
   yAxis = new ALine();
   xLabel = new AStringShape ();
   yLabel = new AStringShape ();
   setAxesLength(theAxesLength);
 }
```

This approach has less code duplication though it does temporarily create inconsistent axes and labels. It is possible but unlikely the end-user would see this inconsistency. This examples also shows that in general, it is better to invoke setter methods to initialize variables rather than set them directly, since they automatically initialize variables that depend on the one being initialized.

Stored vs. Computed Object Properties

Yet another approach to remove code duplication is to not do any dependency calculation in either the constructor or the setters of the dependent properties. Instead, we can take the approach in ABMISpreadshteet and other classes of not storing dependent - we can create new dependent objects each time they are required.

The constructor simply sets the values of the primitive instance variables, without constructing component objects.

```
  public AnInefficientCartesianPlane (int theAxesLength,
                          int theOriginX, int theOriginY ) {
   axesLength = theAxesLength;
   originX = theOriginX;
   originY = theOriginY;
  }
```

The object creation code in the previous constructor is now distributed in the getters. In the previous example, the getters of the dependent properties simply returned the values of the associated instance variables. Now, each of them returns a newly created and correctly initialized object.

```
  public Line getXAxis() {
   return new ALine(toXAxisX(), toXAxisY(), axesLength, 0);
  }
  public Line getYAxis() {
   return new ALine(toYAxisX(), toYAxisY(), 0, axesLength);
  }
  public StringShape getXLabel() {
   return new AStringShape ("X", toXLabelX(), toXLabelY());
  }
  public StringShape getYLabel() {
   return new AStringShape ("Y", toYLabelX(), toYLabelY());
  }
```

The setter simply sets the property value without setting the values of dependent properties:

```
public void setAxesLength(int anAxesLength) {
   axesLength = anAxesLength;
  }
```

If there were multiple independent properties, this approach would be cleaner as we would not have to worry about re-computing dependent objects in each setter. However, this approach is horrendously inefficient as it involves the creation of a new object on each get call. This means additional space overhead as space must be allocated (in the computer "heap") to create the object and the associated display structure ObjectEditor needs for each property. It also means additional time overhead as finding space takes time as does traversing the display structure. With primitive properties, we simply return a value in a slot allocated for each computer call, and none of this overhead occurs. Thus, while computed properties are recommended for dependent primitive values, the opposite is true for dependent object values.

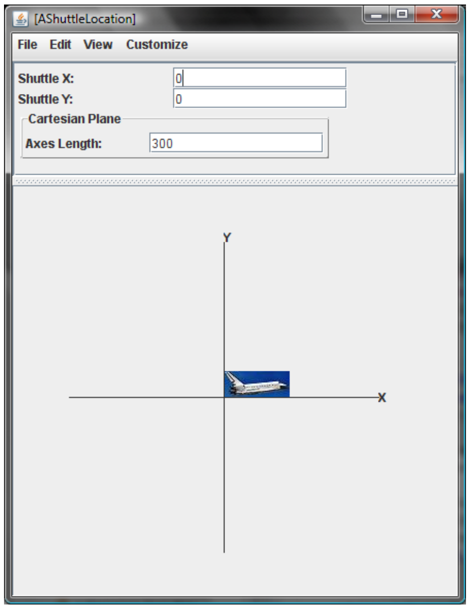
Do not instantiate new objects in getters of object properties.

General Purpose vs. Specialized Shapes

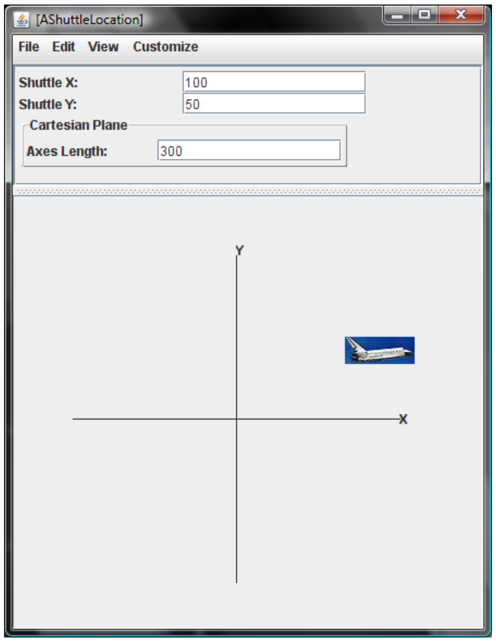
Our approach to stored properties requires the setters of independent properties to be aware of the dependent values. As these dependent values are objects, they can themselves be aware of the dependencies. For example, we can create an X axis whose getter for X computes its value based on the Axeslength property of the Cartesian plane. Similarly, we can create a Y axis whose getter for X computes a different value, again based on the AxeslLength property of the plane. The advantage of this approach is that as we add more dependent properties, we do not have to change the setters for the independent properties. For example, if we add a circle to surround the axes, we do not have to change the setter for Axesength. We can simply add a specialized circle that knows about Axesength and decides how it will react to changes to it. There are at least two disadvantages of this approach. First, we must create a larger variety of component types. For example, we cannot use a generic line for the X or Y axis. We must create specialized versions of these lines for the two axes in which the getters are different. Second, component objects have references to the composite object, that is, have instance variables that hold pointers to the composite object. For instance, the X axis has a reference to the Cartesian plane object so that it can determine the current value of the axes length property. This means both kinds of objects have references to each other, which, as we will see later, can cause overcomable problems when these data structures are displayed. Verall, this is considered an "elegant" approach and an example of *constraint based programming* in which the programmer addresses dependencies by writing functions, such as the getters in this example, rather than procedures, such as setter for the Axeslength property.

Plotted Shuttle

To better understand object composition, let us take a more complex example, shown in Figure ???, which plots the position of an image representation of a (space) shuttle in a 2-D Cartesian plane. The object allows the user to edit the Cartesian coordinates of the shuttle to reposition the shuttle image, as shown in Figure ???. As in the previous example, the user can also change the length of the axes of the Cartesian plane in which the shuttle is displayed.



Shuttle X: 0
Shuttle Y: 0
Cartesian Plane
Axes Length: 300



Shuttle X: 100
Shuttle Y: 50
Cartesian Plane
Axes Length: 300

Flat vs. Hierarchical Logical Structure

As in the case of AcartesianPlane, there are multiple ways to model the logical structure of this object. One approach is to directly include all of the properties of the Cartesian plane and add to them properties for the shuttle image, ShuttleX and ShuttleY:

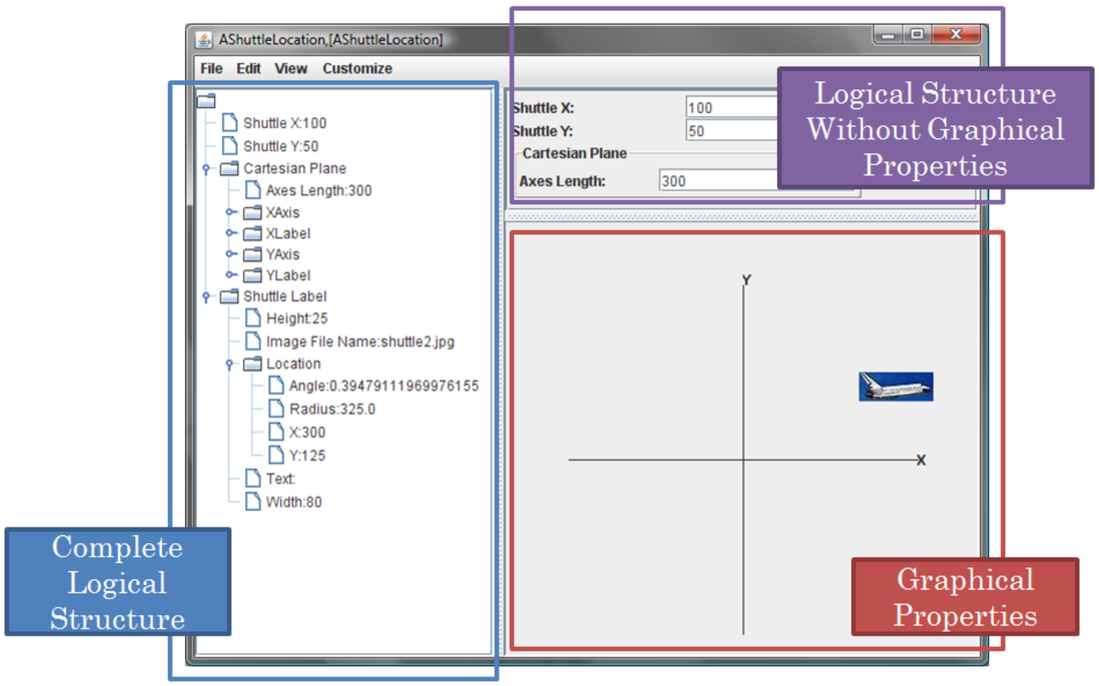
- Editable, Independent Shuttle X(int)
- Editable, Independent Shuttle Y (int)
- Editable, Dependent Shuttle (Image)
- Editable, Independent AxesLength (int) (From Cartesian Plane)

- Readonly, Dependent XAxis (Line) (From Cartesian Plane)
- Readonly, Dependent YAxis (Line) (From Cartesian Plane)
- Readonly, Dependent XLabel (StringShape) (From Cartesian Plane)
- Readonly, Dependent YLabel (StringShape) (From Cartesian Plane)

This is the approach that many students first suggest. However, it requires us to rewrite all of the code for the Cartesian plane. When we study inheritance, we will see how this duplication can be avoided while keeping these properties. However, even without inheritance, it is possible to reuse the code we wrote for the Cartesian plane by simply making the Cartesian plane a property of the plotted shuttle object. Thus, instead of creating a flat structure in which the properties borrowed from the Cartesian plane are direct descendants of the plotted shuttle object, we create a deeper structure, in which these properties are children of the Cartesian Plane, which is a child of the plotted shuttle object:

- Editable, Independent Shuttle X(int)
- Editable, Independent Shuttle Y (int)
- Editable, Dependent Shuttle (ImageWithHeight)
- Readonly, Cartedian Plane (CartesianPlane)

The tree view in Figure ??? shows the logical structure defined by these four properties. The property of type `CartesianPlane` defines the Cartesian space, and the property of type `ImageWithHeight` defines the shuttle image. The CartesianPlane property is readonly in that we do not assign a new Cartesian plane to the plotted shuttle object. However, it is possible to change the editable properties of the plane.


Logical Structure Without Graphical Properties
Complete Logical Structure
Graphical Properties
File Edit View Customize
Y
X

AShuttleLocation,[AShuttleLocation]

Shuttle X:100
Shuttle Y:50
Cartesian Plane
Axes Length:300
XAxis
XLabel
YAxis
YLabel
Shuttle Label
Height:25
Image File Name:shuttle2.jpg
Location
Angle:0.3947911196976155
Radius:325.0
X:300
Y:125
Text:
Width:80

Shuttle X: 100
Shuttle Y: 50
Cartesian Plane
Axes Length: 300

Translating between Coordinate Systems

This example further highlights the difference between the mathematical Cartesian coordinate system and the Java coordinate system. More important, it illustrates the notion of translation between coordinate systems, which is often done in graphics applications.

The ShuttleX and ShuttleY properties specify the X and Y coordinates in the true mathematical Cartesian space defined by the `CartesianPlane` property rather than the X and Y coordinates in the inverted Java space. Moreover, they specify the coordinates of the lower left rather than the upper left corner of the shuttle image. For example, when the shuttle is at position 0, 0, its lower left corner is at the origin of the Cartesian plane (Figure ???). Thus, when we set the shuttle X and Y coordinates to (100, 100), we are saying that the location of the lower left of the shuttle image is at a distance of 100 (pixels) above and to the right of the origin of the Cartesian Plane drawn in the graphics window and not that the position of the upper left corner is at a distance of 100 below and to the right of the upper left corner of the graphics window. Because the Java Coordinates of the origin of the CartesianPlane are (200, 200), the Java coordinates of the lower left corner are (200 + 100, 200-100. Thus, our new class must translate between the two coordinate systems when repositioning the shuttle image. The functions windowX) and windowY) convert the ShuttleX and ShuttleY positions to appropriate Java X and Y coordinates, respectively:

```
int toWindowX() {return ORIGIN_X + shuttleX;}
int toWindowY() {return ORIGIN_Y - shuttleY - shuttleImage.getHeight();}
```

The height of the image is subtracted from `ORIGIN_Y - shuttle` to account for the fact that ObjectEditor uses the upper left corner of the image as its location

These two functions are called by the setters of the two properties to set the coordinates of the shuttle image:

```
public void setShuttleX(int newVal) {
   shuttleX = newVal;
   shuttleImage.setX(toWindowX());
 }
 public void setShuttleY(int newVal) {
   shuttleY = newVal;
   shuttleImage.setY(toWindowY());
 }
}
```

These are pure functions that do not access global variables of the class. As a result, they could have been created also as static functions.
The class defines the expected instance variables and getters/setters for the four properties.

```
public class APlottedShuttle implements PlottedShuttle {
 static final String SHUTTLE_IMAGE_FILE_NAME = "shuttle2.jpg";
 static final int ORIGIN_X = 200, ORIGIN_Y = 200;
 static final int AXES_LENGTH = 300;
 int shuttleX = 0, shuttleY = 0;
 CartesianPlane cartesianPlane;
 ImageWithHeight shuttleImage;
 public APlottedShuttle(int anX, int aY) {
   cartesianPlane = new ACartesianPlane (AXES_LENGTH,
                                   ORIGIN_X, ORIGIN_Y);
  shuttleImage = new AnImageWithHeight(SHUTTLE_IMAGE_FILE_NAME);
  setShuttleX(anX);
  setShuttleY(aY);
 }
```

The constructor initializes the two object instance variables by creating appropriate instances of `ACartesianPlane` and `AShuttle`. The actual parameters of the constructors of these classes are based on values of named constants. These define the origin and axes of the CartesianPlane property. The coordinate translation functions are called both by the constructor and the two setters – as in the previous example we can create an inconsistent component and then make it consistent by calling the setters.

Computing the Height of an Image

As we see above, to correctly do the translation, we need the height of the image. As we do not want to clip the image, we would like Java to draw the entire image but tell us its height. The class of the shuttle image illustrates how this is done:

```
public class AnImageWithHeight implements ImageWithHeight {
 int x, y;
 String imageFileName;
 int imageHeight;
 public AnImageWithHeight(String anImageFileName) {
  imageFileName = anImageFileName;
  Icon icon = new ImageIcon(imageFileName);
  imageHeight = icon.getIconHeight();
 }
 public int getX() {return x;}
 public void setX(int newX) {x = newX;}
 public int getY() { return y; }
 public void setY(int newY) {y = newY;}
 public String getImageFileName() {return imageFileName;}
 public int getHeight() { return  imageHeight;}
}
```

The class defines the X, Y, and ImageFileName properties required by ObjectEditor to draw an instance of it as an image. In addition, it provides the optional, Height property, which returns the height of the image stored in the file. The Icon and ImageIcon classes provided by the javax.Swing package provide the necessary tools to determine this height, as shown above. Both ImageFileName and Height are readonly as the displayed image is not expected to change in this example.

Object Tree View, Main View and Drawing View

The displays created in Figure ??? and ??? illustrate the general approach ObjectEditor takes to display an object. It creates three kinds of views, tree view, main view, and drawing view, each of which can be hidden or displayed using the ObjectEditor API or interactive commands. The complete logical structure of the object is displayed in the tree view. All atomic shape nodes in the structure are displayed in the drawing view. The remaining nodes are displayed in the main (text) view. Thus, in Figure ???, the ShuttleImage property of PlottedShuttle is shown in the tree and graphics views but not the main view. Similarly, the ShuttleX property of this object is shown in the tree and main views but not the graphics views. The Cartesian plane property has some components that are shapes and some that are not. The former are displayed in the drawing (and tree) windows and the latter in the main windows. The non-shape components are nested properly in the main window to show that they are components of Cartesian plane rather than the plotted shuttle. Thus, like the tree view, the main view shows the hierarchy, though in a different way. As this view consumes more space, it omits the atomic shapes from the displayed structure.

If a node has only shape descendents, it is not displayed in either the main or graphics view. For example, if the Cartesian plane did not have the AxesLength property, no display for it would have been created in the main window in either Fiugure ??? or Figure ???.

The ObjectEditor tree view is useful to help debug translation and other shape placement functions, as it shows the exact coordinates of each shape. In addition, hovering over a shape in the graphics view shows its coordinates. Like the main view, the tree view can be edited to call the setters of the displayed properties. The main view provides a more convenient user-interface to so, but of course, cannot be used to change editable properties of shapes, as these are not displayed in it.

Creating Arbitrary Shapes

We have seen in this section that it is possible to create complex geometric objects such as `CartesianPlane` and `PlottedShuttle` from atomic shapes ones such as `Line` and `ShuttleImage`. In fact, we can create arbitrary computer graphics from atomic shapes. In particular, from lines it is possible to create triangles, and from triangles it is possible to create arbitrary computer graphics. However, you will need to take a course on computer graphics to see how this is exactly done. Figure ???? gives the intuition behind this idea.

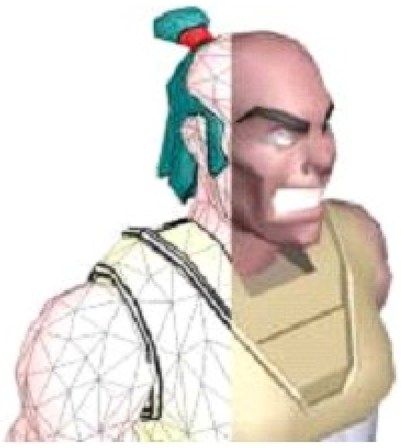


Figure 1. Composing triangles to create a complex shape

Exercises

1. Distinguish between read-only, editable, stored and computed properties.

2. Assume two implementations `DollarMoney` and `PoundsMoney`, of the following interface:

```
public interface Money {
      public int getPounds();
      public int getDollars();
}
```

DollarMoney (PoundMoney) has a constructor to initialize the money to a dollar (pound) amount; and the two methods above return this amount in pounds and dollars, respectively. Consider the following class, which is meant to represent the total money a person has in UK and USA. Identify and correct the errors and violations of style rules covered in this chapter. Also, classify the methods in the class into polymorphic and non-polymorphic, and overloaded and non-overloaded.

```
public class UK_USA_Assets implements Money {
      static PoundMoney assets;
      public UK_USA_Assets (DollarMoney usaAssets, PoundMoney ukAssets) {
            assets = add (usaAssets, ukAssets);
      }
      public DollarMoney add (DollarMoney arg1, DollarMoney arg2)  {
            return new DollarMoney (arg1.getDollars() + arg2.getDollars());
      }
      public DollarMoney add (DollarMoney arg1, PoundMoney arg2)  {
            return new DollarMoney (arg1.getDollars() + arg2.getDollars());
      }
      public DollarMoney subtract (Money arg1, Money arg2)  {
            return new DollarMoney (arg1.getDollars() + arg2.getDollars());
      }
      public static int getPounds() {
            return assets.getPounds();
      }
      public int getDollars() {
            return assets.getDollars();
      }
}
```

3. Extend the temperature interface of Chapter 4, problem 9, so that the Fahrenheit property is editable rather than read-only.

4. Implement the interface in a class, `ACentigradeTemperature`, that represents the temperature as a centigrade value and provides a constructor to initialize it to a value specified in centigrade. Thus:

   ```
   new ACentigradeTemperature(100)
   ```

   returns a new temperature representing 100 degree centigrade.

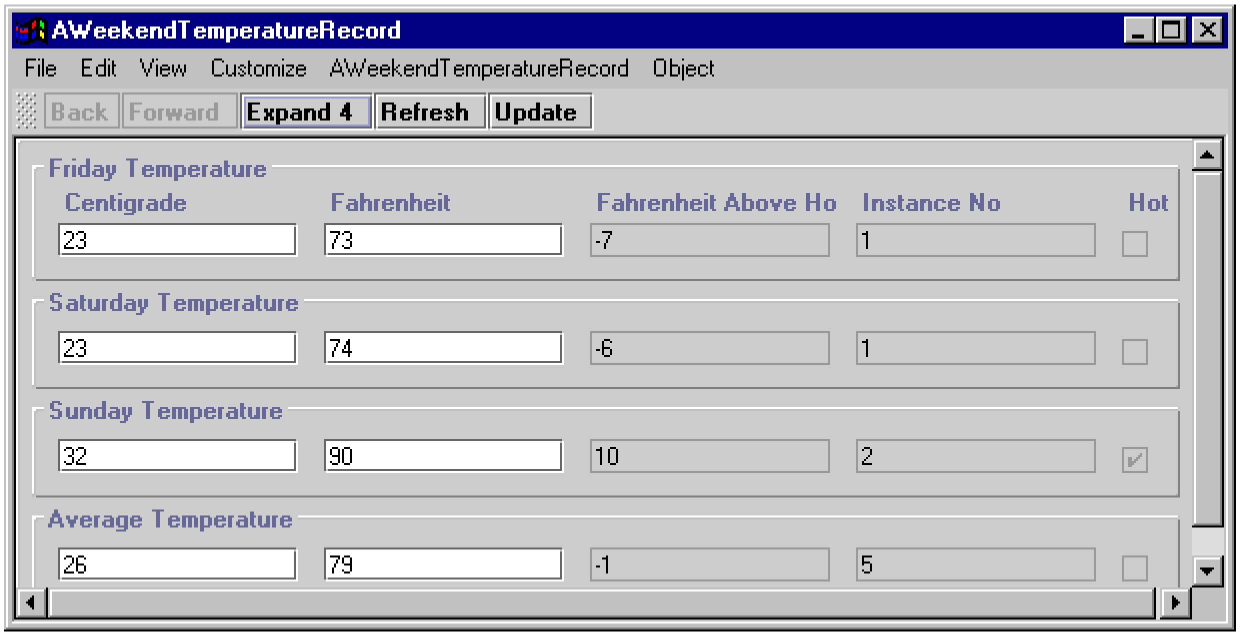
5. Provide another implementation of the interface that represents the temperature as a Fahrenheit value and provides a constructor to initialize it to a value specified in centigrade. Thus:

   ```
   new AFahrenheitTemperature (212)
   ```

   returns a new temperature representing 212 degree Fahrenheit.

6. Implement an object, displayed below, defining the temperatures recorded during some weekend. It has four properties: the first three specify the temperatures recorded on the three

days of the weekend, Friday, Saturday, and Sunday; and the fourth displays the average of the temperatures of these three days.



	Centigrade	Fahrenheit	Fahrenheit Above Ho	Instance No	Hot
Friday Temperature	23	73	-7	1	☐
Saturday Temperature	23	74	-6	1	☐
Sunday Temperature	32	90	10	2	✓
Average Temperature	26	79	-1	5	☐

You will not (by default) get a tabular display of the kind shown above; instead you will get each item displayed on a separate line, which is acceptable. The display has been customized so that it takes less space.

It is up to you which representation you use for a particular temperature, but you should use instances of both `ACentigradeTemperature` and `AFahrenheitTemperature` to represent the temperatures. That is, one to three of the four temperatures should be represented using instances of `ACentigradeTemperature` and the remaining should be represented using instances of `AFahrenheitTemperature`. (Can you tell from the figure that Friday and Saturday temperatures use different representation?)

It is also up to you whether you compute the average of the three temperatures inside in the temperature classes (`ACentigradeTemperature` & `AFahrenheitTemperature`) or in the user of these classes (`AWeekendTemperatureRecord`).

7. Draw diagrams giving the physical and logical structure of the object above.

8. What are the consequences, on the users of `AWeekendTemperature`, of using a particular representation of a temperature?