COMP 110
Prasun Dewan[1]

7. Representation

Here we take object-based programming to the next level by gaining more experience in defining objects and learning some new concepts. Using the BMI spreadsheet example, we will learn a test-first approach in which we first create test code before we complete the objects we wish to test. This will lead the idea of a method stub – a method with no body that has the correct header.

We will define new kinds of objects that capture graphical shapes such as points and lines. We will use them to gain more experience with multiple implementations of the same interface. We will see that the representation can determine not only the efficiency of our code but also its correctness. These objects will also serve as examples illustrating the difference between coordinate systems supported in Mathematics and computer graphics. We will then look at the idea of class methods and variables, which allow us to invoke operations on a class. In addition, we will see objects whose instance variables and properties are objects rather than double, int, and other primitive values we have seen so far. Thus, we will see that an object can be made the property/instance variable of another object, thereby supporting arbitrary hierarchical structures. This will lead us to the distinction between structured and atomic types. We will see that object variables, unlike primitive ones, are initialized to special **null** values, which will make it even more important to initialize them in constructors.

We will also see new kinds of objects that go far beyond commercial spreadsheets, by supporting two-way dependencies between displayed items. Finally, we will see the differences between three alternative ways of programming – top-down, bottom-up, and middle-out.

Testing Objects

As programmers, we are very fallible, easily likely to make errors – especially silly ones such as inverting a formula or assigning the correct value to the wrong variable. The crashes of commercial programs we encounter such as the one shown in Figure 1 are evidence that even experienced programmers make mistakes. Therefore, it is important to put significant effort in testing the objects we write for absence of errors.

To illustrate the nature of the testing process, let us return to the example of the BMI spreadsheet. Figure 2 gives published values of BMI for different values of weight and height.

---

[1] © Copyright Prasun Dewan, 2000.

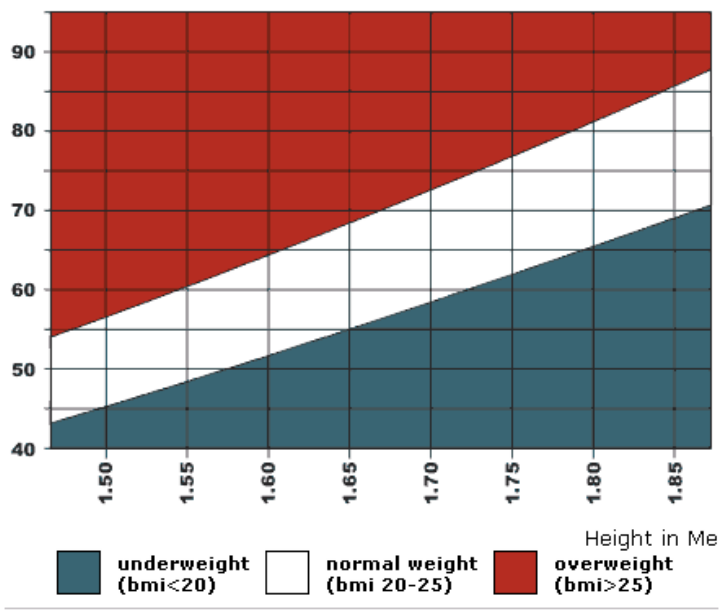**Figure 1. Example of a real-world program error**

Figure 2. Published BMI values

It is possible to test our spreadsheet example by manually comparing the BMI values computed by our spreadsheet with the published ones. Figure 3illustrates a better approach, in which this comparison is done by a special class, `ABMISpreadsheetTester`.

The class defines a special method, invoked using the "Test(double, double, double)" menu item (Figure 3 top-left), which takes three parameters defining a correct height, weight, and BMI triple (Figure 3 top-right). It then determines the BMI value yielded by the BMI spreadsheet for the height and weight, and reports the difference between the computed and correct value as the error (Figure 3 bottom). In this
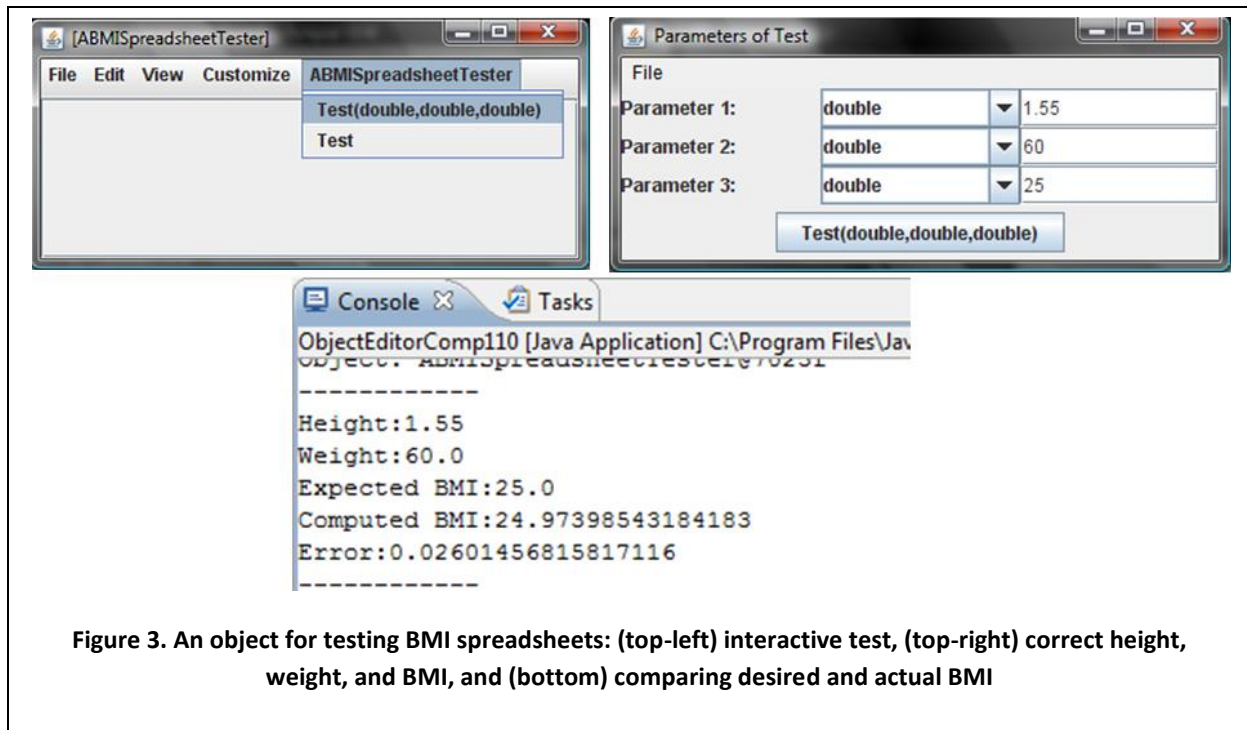
Figure 3. An object for testing BMI spreadsheets: (top-left) interactive test, (top-right) correct height, weight, and BMI, and (bottom) comparing desired and actual BMI

example, the error is so small that it is insignificant. It is an artifact of the fact that our chart reports approximate rather than exact BMI values. If a few other triplets yield this small an "error," we know we have used the correct formula in our implementation.

The following code gives the implementation of the method:

```java
public class ABMISpreadsheetTester {
    public void test (
            double theHeight, double theWeight, double theCorrectBMI) {
        BMISpreadsheet bmiSpreadsheet =
                new ABMISpreadsheet(theHeight, theWeight);
        double computedBMI = bmiSpreadsheet.getBMI();
        System.out.println("------------");
        System.out.println("Height:" + theHeight);
        System.out.println("Weight:" + theWeight);
        System.out.println("Expected BMI:" + theCorrectBMI);
        System.out.println("Computed BMI:" + computedBMI);
        System.out.println("Error:" + (theCorrectBMI - computedBMI));
        System.out.println("------------");
    }
}
```

The implementation uses concepts we have learned before. It first creates an instance of `ABMISpreadsheet`. It passes to the constructor of `ABMISpreadsheet`, the values of its height and weight parameters. It then calls the getter method in the instance, and stores the return value in the variable, `computedBMI`. Finally it prints the values of its `height` and `weight` parameters and the

difference between the correct BMI values, passed to it as a parameter, and the BMI computed by `ABMISpreadsheet`.

The following is an alternative implementation in which the parameterless constructor is used to create the instance of `ABMISpreadsheet`. As the height and weight are not passed as constructor parameters, setter methods are used to initialize the two properties. This time we do pass parameters, which determine the new values of instance variables.

The basic idea, used in both implementations, is to initialize the variables storing the independent properties, height and weight, either using a constructor or setter methods, and then to retrieve the computed property to determine the error, if any.

```java
public class ABMISpreadsheetTester {
    public void test (
            double theHeight, double theWeight, double theCorrectBMI) {
        BMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet();
        bmiSpreadsheet.setHeight(theHeight);
        bmiSpreadsheet.setWeight(theWeight);
        double computedBMI = bmiSpreadsheet.getBMI();
        System.out.println("------------");
        System.out.println("Height:" + theHeight);
        System.out.println("Weight:" + theWeight);
        System.out.println("Expected BMI:" + theCorrectBMI);
        System.out.println("Computed BMI:" + computedBMI);
        System.out.println("Error:" + (theCorrectBMI - computedBMI));
        System.out.println("------------");
    }
}
```

The above testing process, while automating error correction, is still fairly labor intensive. Every time we create a new version of the spreadsheet, we must redo the process of manually entering the `height`, `weight`, and `BMI` parameters of the test method. The other (overloaded) test method of `ABMISpreadsheetTester` fixes this problem.

As we see from the output, the method automatically checks three different triplets for correctness (Figure 4). As a result, we don't have to enter these values each time we wish to test the spreadsheet.

The implementation of this test method is trivial. The three triplets are hardwired in the method. It simply calls the other test method, defined by the same class, for each triplet.

```java
public void test ()
    test (1.65, 55, 20.0);
    test (1.55, 60, 25);
    test (1.80, 65, 20);
}
```
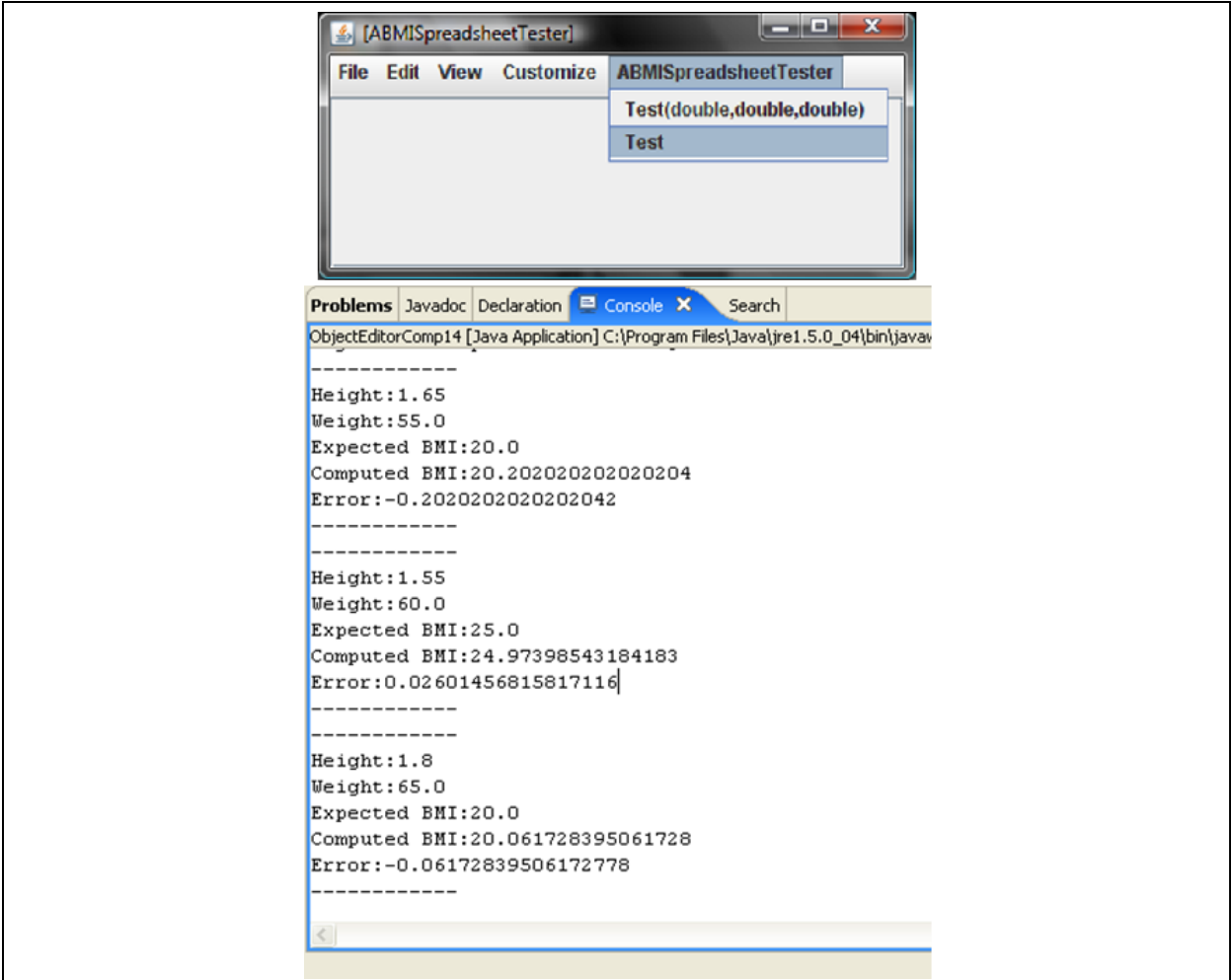
Figure 4. Calling the test method which tests BMI Spreadsheet with three different sets of values

Tester-Last vs. Tester-First

The testing approach used above is illustrated in Figure 5.

Figure 5 (left) depicts this approach abstractly, and Figure 5 (right) illustrates it concretely using the BMI spreadsheet example. In this approach, we first create the complete code to be tested, such as the class `ABMISpreadsheet`, and then the program that tests it, such as `ABMISpreadsheetTester`. If we find any bugs, we change the tested code, and then redo the test, possibly enhancing the tester. It is traditional to create the tested code before the tester – after all what can we do with the tester if the tested code is not ready?

This approach indeed makes sense if our goal is to test all aspects of the tested code together. However, it is often more convenient to incrementally test different aspects of the code as we write them. This approach allows us to more easily pinpoint the cause of bugs. It might seem that tested code and testing code should evolve together. For example, it does not seem to make sense to put code in the tester to
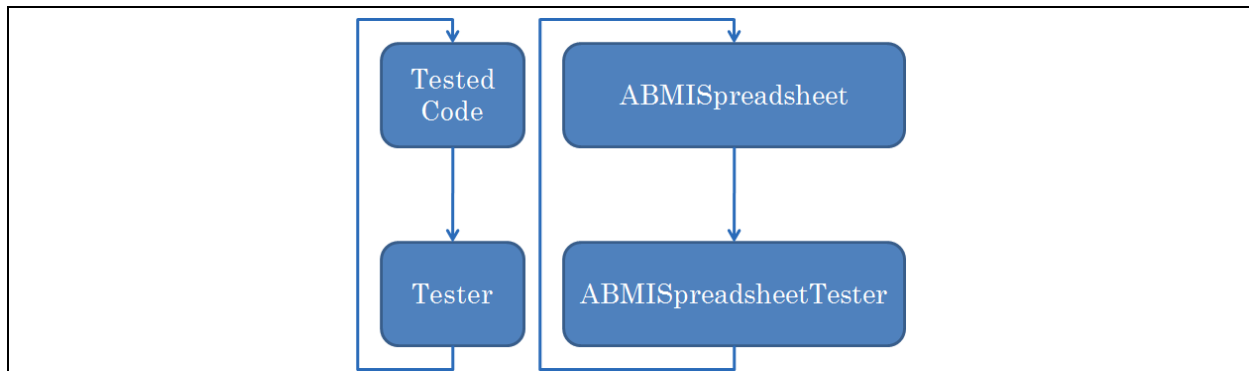
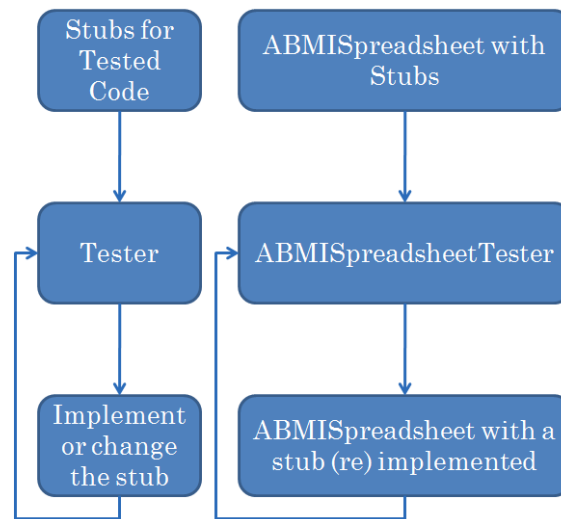**Figure 5. The traditional test-last approach**

Figure 6. The test-first (stub-based) approach

check the value of `getBMI` until the `getBMI` method has been written. In fact, if we do so, we will get compile errors. For example, the following code in our tester:

```
double computedBMI = bmiSpreadsheet.getBMI();
```

will not compile if the `getBMI` method has not been implemented.

The new approach, on the other hand, requires us to write the complete tester before we start "real coding" of the class to be tested. The idea is to write "stubs" for the various public methods implemented by the class. A stub is a skeleton of the actual method that has the correct header but has a body that does nothing except perhaps return some arbitrary value in case the method is a function. For example, the stub for `getBMI` could be:

```
public double getBMI() {
        return 0;
}
```

Once we have written the stubs, we can write and compile our tester. After this, each stub can be incrementally replaced with real code and then tested. Figure 6 shows this approach both abstractly and using a concrete example.

Before we explain why the tester should be written first, let us use the example of BMI spreadsheet to illustrate its details. Under this approach, we will first write the complete interface of the class to be tested:

```
public interface BMISpreadsheet {
      public double getHeight() ;
      public void setHeight (double newHeight) ;
      public double getWeight() ;
      public void setWeight(double newWeight) ;
      public double getBMI() ;
}
```

Next we write stubs for each of these methods, returning arbitrary values in the case of functions:

```
public class ABMISpreadsheet implements BMISpreadsheet {
      public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {}
      public ABMISpreadsheet() {}
      public double getHeight() {return 0;}
      public void setHeight(double newHeight) { }
      public double getWeight() {return 0;}
      public void setWeight(double newWeight) {}
      public double getBMI() {return 0;}
}
```

The stub-based implementation does not declare instance variables. These will be created when we get to the nitty-gritty of the implementation and decide how to represent the object. We can postpone creating these variables as stubs access no variables.

We are now ready to write the complete class, ABMISpreadsheetTester. It will compile correctly because of the stubs. Of course, these stubs don't do anything real, as shown in Figure 7. No matter what the height and weight passed to the constructor or the setters, all three properties remain 0.

If we run the tester at this point, we will get significant errors, as shown in Figure 8. These errors concretely tell us what the correctness criteria are. Recall that an interface is not a sufficient mechanism for specifying the behavior of a classes. For example, ABMISpreadsheet correctly implements the desired interface. It is possible to more fully define the semantics of a class by writing a tester for it. As the specification should come before the implementation, it does indeed make sense to write the tester before the tested code. Thus, in this approach we change the tester when we refine a specification but not when we implement the specification.
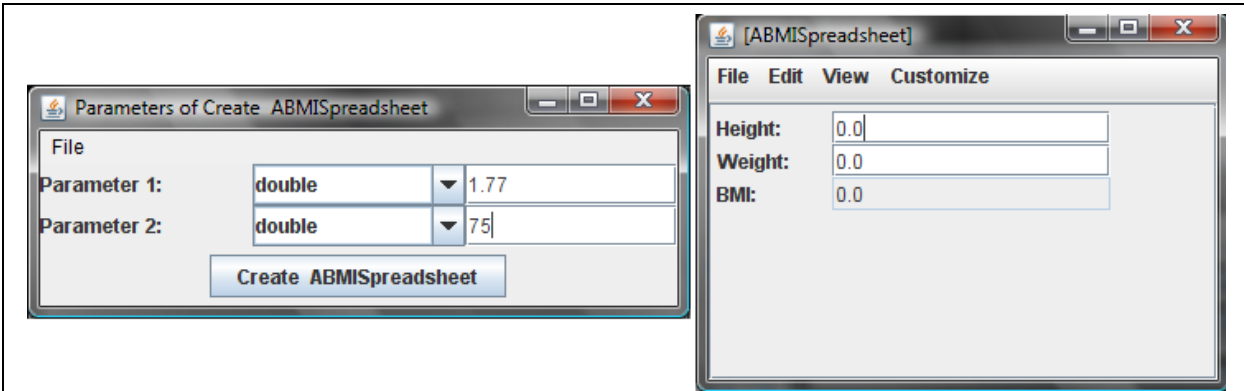
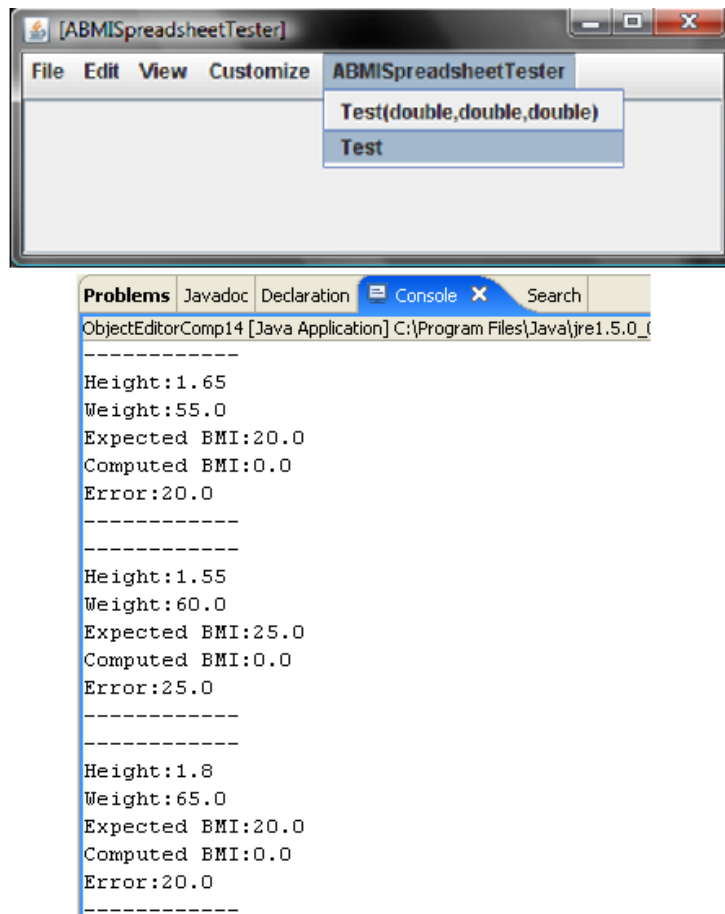**Figure 7. Return values independent of values of constructor parameters**

Figure 8. Errors denote the expected semantics

Null Pointers

Consider again the testing code we wrote. Suppose we did not initialize the two variables, `bmiSpreadsheet` and `computedBMI`, as shown below:

```
public class ABMISpreadsheetTester {
    public void test (
            double theHeight, double theWeight, double theCorrectBMI) {
            BMISpreadsheet bmiSpreadsheet; // uninitialized object variable
            double computedBMI ; // uninitialized primitive variable
            System.out.println("------------");
            System.out.println("Height:" + theHeight);
            System.out.println("Weight:" + theWeight);
            System.out.println("Expected BMI:" + theCorrectBMI);
            System.out.println("Computed BMI:" + computedBMI);
            System.out.println("Error:" + (theCorrectBMI - computedBMI));
            System.out.println("------------");
    }
}
```

If we do not initialize a primitive variable, that is a variable whose type is one of the primitive types we studied in the previous chapter, Java stores an appropriate default value for it. For example, the default value of the primitive variable `computedBMI` is 0.0. On the other hand, if we do not initialize an object variable, that is a variable whose type is an object type (class or interface), Java does *not* store a default legal object of that type. Instead, it stores a special value, called the *null value*, which signifies the absence of an object value. A variable storing this value is called a *null pointer,* since it does not refer to a legal object in memory.

If we try invoke some object operation on a null object:

<p align="center"><code>bmiSpreadsheet.getBMI()</code></p>

we will get an error message, indicating a `NullPointerException`, essentially saying that the variable does not store any object. Beware of this error - it is very easy to write programs that invoke operations on uninitialized object variables.

Point

To gain more experience with objects, let us define a new class of objects – one that defines points in a two-dimensional space. This is an important type, forming the basis for most graphics/geographic applications.

As you may remember from Math, a point has an angle (θ), radius (R), X, and Y coordinate (Figure 9 left). The angle and radius of a point are called its Polar coordinates and the X and Y coordinates are called its Cartesian coordinates. Each of these pairs of coordinates completely represents the point, and it is possible to convert one representation to the other. A user of a point object may be interested in both sets of coordinates. Consider a rocket we have launched – we may be interested in interested in statistics about the highest point of the rocket relative to the origin (launching point): the angle a
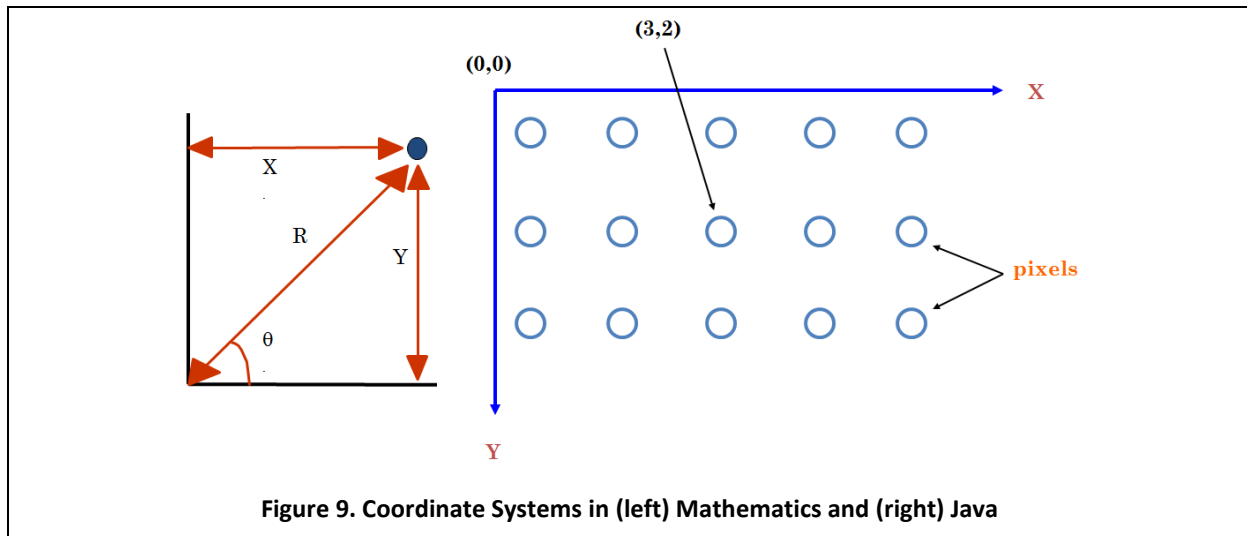
Figure 9. Coordinate Systems in (left) Mathematics and (right) Java

straight line to it from the origin makes with the horizontal axis, and its total, horizontal, and vertical distance from the origin. Thus, we need the value of four of its coordinates.

As it turns out, the coordinate system familiar to us from Mathematics does not apply directly to computer graphics. In the Mathematics coordinate system, we can specify continuous values along the X and Y dimensions, locating a point with arbitrary precision. As we saw earlier, computer numbers cannot model real numbers precisely. More importantly, the computer cannot draw at arbitrary locations on the screen. It can draw only at a subset of the infinite points on the screen called the screen pixels (Figure 9 right). These are the points the hardware drawing gun colors to draw on the screen. The pixels are equally spaced in both the X and Y directions. The pixel spacing or pixel unit determines the resolution of the screen – the higher the resolution, the smaller the pixel unit. If the pixel spacing is p (in some unit of length such as millimeters) on a screen with width W and height H, then the resolution of the screen is W/p by H/p.

The coordinates of a point are given in pixel units. Thus, the Cartesian coordinates (x, y) indicates a pixel that is x pixel units from the origin in the X direction and y pixel units from the origin in the Y direction. Since these coordinates refer to discrete rather than continuous points on the screen, they are specified as int values. Because they are specified in pixel units, the actual location of a point, (x,y) depends on the resolution of the screen. Thus, if on a particular screen, the point (256, 174), is at a location 10cm from the Y axis and 8 cm from the X axis, then on a screen of the same size with twice the resolution, it will be 5 cm from the Y axis and 4 cm from the X axis.

Each window is associated with its own coordinate system so that we can create shapes independently in different windows. The origin of the coordinate system is the upper left corner of the window – X values increase from left to right, and Y values increase from up to down and not down to up as we are used to from Mathematics (Figure 9 (right)). The reason for not following the Mathematics convention is that it is usual to think of the upper left corner of a window as its position and specify positive coordinates, relative to this corner, for points within the window.

We are now ready to define an interface for the Point object:

```java
public interface Point {
    public int getX();
    public int getY();
    public double getAngle();
    public double getRadius();
}
```

For the reasons given above, the getters for the Cartesian coordinates return int values. The Polar coordinates, on the other hand, are double values, as computer graphics do not constrain them to be whole numbers. In fact, the angle would be a decimal number between 0 and 2π, and thus cannot be usefully approximated by int values.

In this interface, we have not defined any setters because we are considering a point object to be unchangeable or *immutable.* Instead of invoking setters to create a new point out of an existing point, we can instantiate a new Point object. For example, as the rocket moves, we can create a new Point object for each of its positions in which we are interested, rather than continuously changing a single Point object.

Our next step is to create a class for implementing the interface above. As mentioned above, a point can be represented in the implementation using either Cartesian or Polar Coordinates. Let us choose the former. Following the tester-first policy, let us first write the stubs for the class:

```java
public class ACartesianPoint implements Point {
    public ACartesianPoint(int theX, int theY) {}
    public int getX() {return 0;}
    public int getY() {return 0;}
    public double getAngle() {return 0;}
    public double getRadius() {return 0;}
}
```

There are many cases we can try in the tester for this class. For now, let us just choose two simple ones:

```java
public class ACartesianPointTester {
    public void test (
                int theX, int theY,
                double theCorrectRadius, double theCorrectAngle) {
        Point point = new ACartesianPoint(theX, theY);
        double computedRadius = point.getRadius();
        double computedAngle = point.getAngle();
        System.out.println("------------");
        System.out.println("X:" + theX);
        System.out.println("Y:" + theY);
        System.out.println("Expected Radius:" + theCorrectRadius);
        System.out.println("Computed Radius:" + computedRadius);
        System.out.println(
            "Radius Error:" + (theCorrectRadius - computedRadius));
        System.out.println("Expected Angle:" + theCorrectAngle);
        System.out.println("Computed Angle:" + computedAngle);
        System.out.println(
            "Angle Error:" + (theCorrectAngle - computedAngle));
        System.out.println("------------");
    }
    public void test () {
        test(10, 0, 10, 0); // 0 degree angle
        test(0, 10, 10, Math.PI / 2); // 90 degree angle
    }
}
```

As we have chosen the Cartesian representation, we must be able to convert from Cartesian coordinates to Polar coordinates. From Figure 9 we can see that:

$$R = sqrt\ (X^2 + Y^2)$$
$$\theta = arctan\ (Y/X)$$

We are now ready to implement the complete class:

```java
public class ACartesianPoint implements Point {
    int x, y;
    public ACartesianPoint(int theX, int theY) {
        x = theX;
        y = theY;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public double getAngle() { return Math.atan((double) y/x); }
    public double getRadius() { return Math.sqrt(x*x + y*y); }
}
```

Polar Representation

Even though pixels are arranged on the X and Y axes, there is no reason we need to choose Cartesian coordinates for representing point objects. It is equally valid to create an implementation based on Polar coordinates. This time, we must know how to convert from Polar to Cartesian coordinates:

$$X = radius*\cos(\theta);$$
$$Y = radius*\sin(\theta);$$

The Polar implementation is analogous to the Cartesian one:

```java
public class APolarPoint implements Point {
    double radius, angle;
    public APolarPoint(int theRadius, int theAngle) {
        radius = theRadius;
        angle = theAngle;
    }
    public int getX() { return (int) (radius*Math.cos(angle)); }
    public int getY() { return (int) (radius*Math.sin(angle)); }
    public double getAngle() { return angle; }
    public double getRadius() { return radius; }
}
```

Thus, we see once again the concept of creating multiple implementations of the same interface. As before, we choose an implementation by invoking the corresponding constructor:

```java
Point point1 = new ACartesianPoint(50, 50);
Point point2 = new APolarPoint(70.5, Math.PI/4);
```

The idea of multiple implementations of an interface is particularly applicable to geometric objects, as these tend to have numerous useful representations.

ObjectEditor Graphics

Let us continue to focus on `ACartesianPoint`. Figure 10 shows how ObjectEditor displays an instance of this class. As we see in the figure, ObjectEditor understands that our object represents a geometric point, and uses the coordinate system used in computer graphics to display it in a special drawing window. This view, however, does not tell us the exact values of the four coordinates. We can execute the View→Tree command to show the values of the four coordinates in a tree view (Figure 11 top-left). This command is a toggle, so executing it again will remove the tree view. If we are interested only in the tree view, we can execute View→Drawing to remove the graphic view (Figure 11 bottom-left). This command is also a toggle.

How did ObjectEditor recognize `ACartesianPoint` as a class that defined geometric points? ObjectEditor assumed an object represents a geometric point if:

- The name of its interface or class has the string "Point" in its name
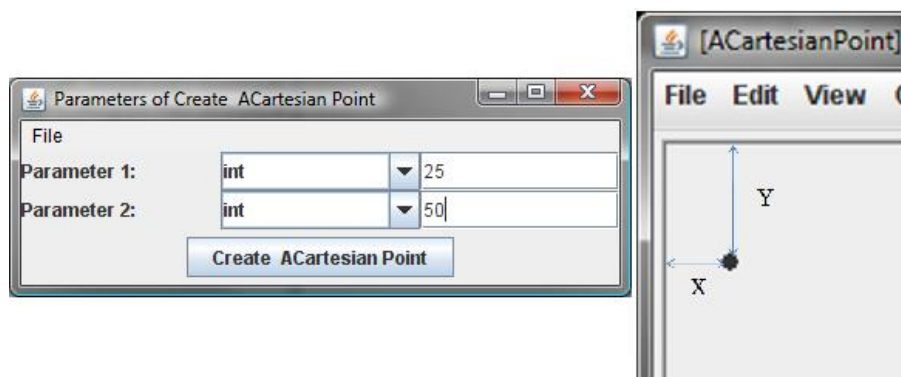- It has int properties, named x and y, that represent its Cartesian coordinates

Figure 10. ObjectEditor graphical representation of a point: (left) instantiating, (right-top) entering X and Y coordinates, and (right-bottom) ObjectEditor window
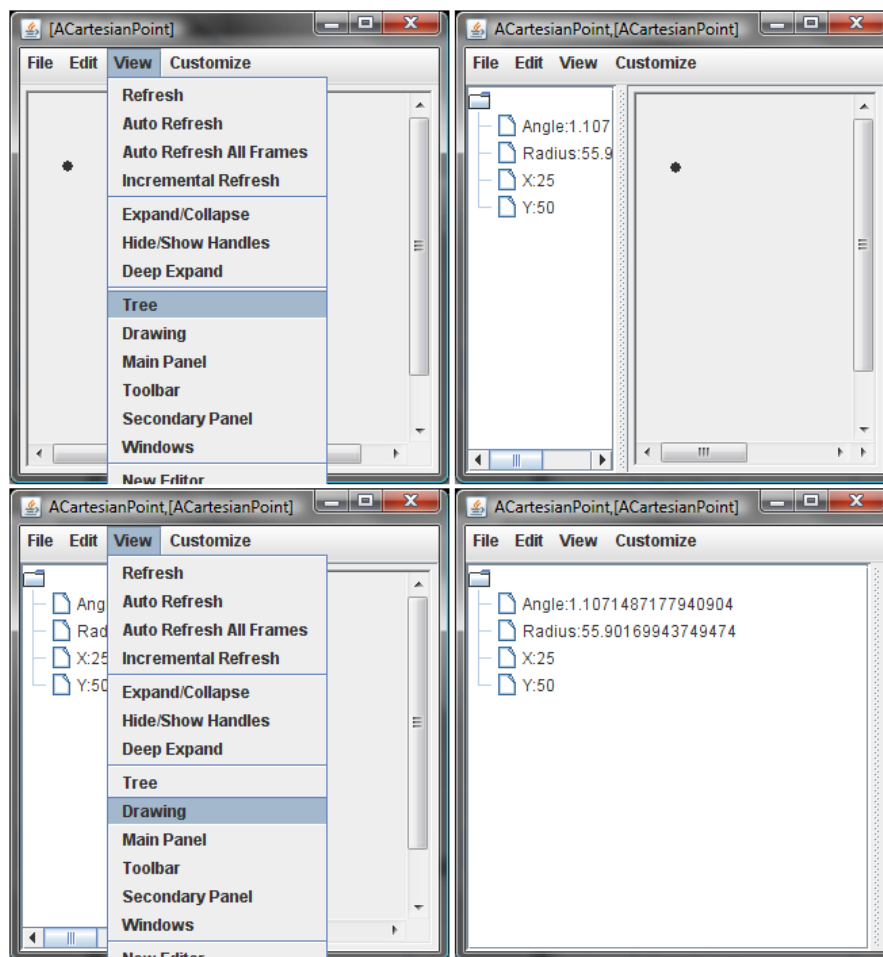

Figure 11. (top-left) default graphics-only view, (top-right) tree and graphics view, (bottom-left) the drawing toggle command, and (bottom-right) the tree-only view

The class can have additional properties such as angle and radius in our implementation – these are not used to construct the graphics view. Thus, these conventions are in the spirit of the JavaBean conventions, which also derive Object characteristics based on the patterns programmers follow in
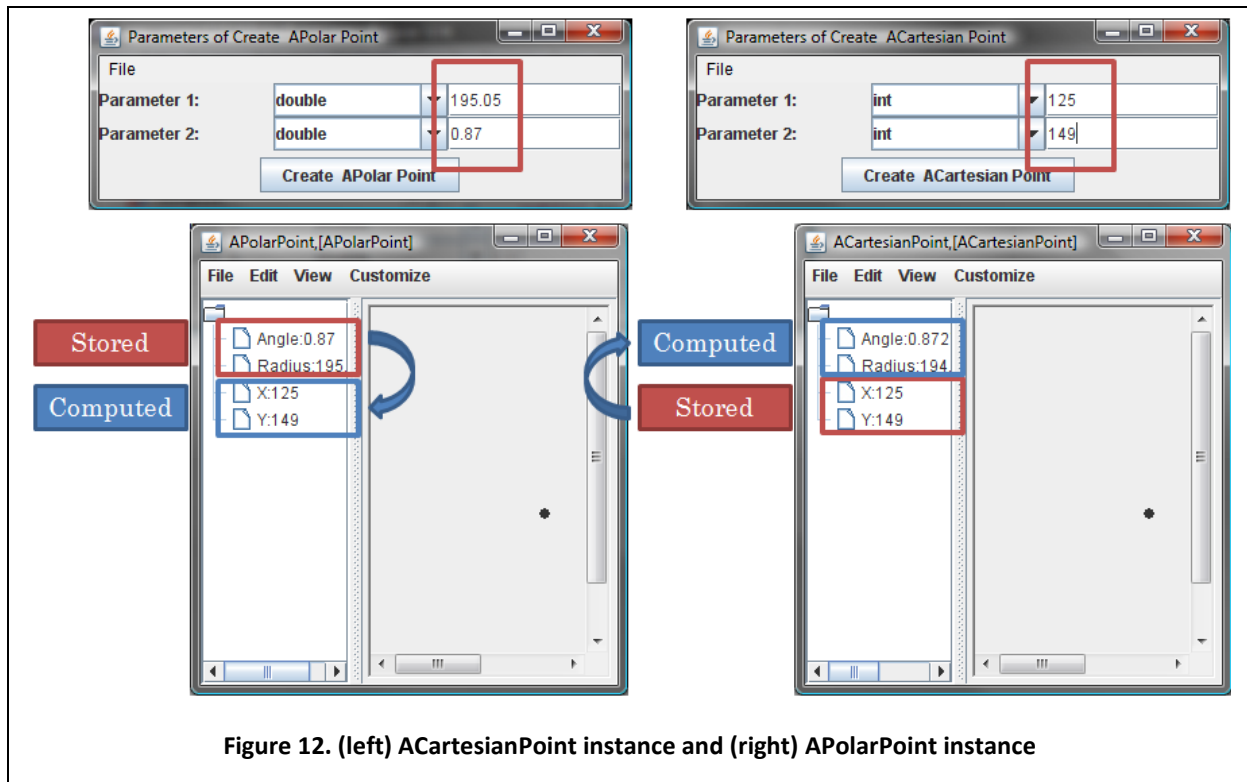
Figure 12. (left) ACartesianPoint instance and (right) APolarPoint instance

coding classes and interfaces. In fact, these conventions are built on top of JavaBean conventions – they assume that the latter are followed in defining the X and Y properties. We will later see conventions for describing lines, rectangles, and text-boxes, which will allow us to easily create graphics applications.

ObjectEditor conventions for a point are also followed by the class `APolarPoint`. Thus, instances of it are displayed like instances of `ACartesianPoint`, as shown in Figure 12.

Conversions Errors

Figure 12 shows something interesting: instances of these two classes representing the "same" Mathematical point are not identical!

The two instances have the same Cartesian coordinates but different Polar coordinates! The reason has to do with the representations used in the two classes. In the case of `APolarPoint`, the Polar coordinates are the stored properties, and the Cartesian coordinates are the computed properties, while in the case of `ACartesianPoint`, the opposite is true. When converting Polar to Cartesian coordinates, a double value is truncated to an int:

```
public int getX() {
        return (int) (radius*Math.cos(angle));
}
```

When the two truncated Cartesian coordinates are converted back to Polar coordinates, we get different values.

Thus, we may choose different representations, not just to tradeoff space for efficiency, but to tradeoff accuracy in one set of properties for accuracy in another set. A corrollary of this discussion is that if we are going to create a single representation, we must choose it wisely since it effects efficiency and nature of conversion errors.

Class Methods

Consider a getter method in `ACartesianPoint`, `getNumberOfInstances`, that counts how many instances of the class have been created so far. Figure 13 illustrates how the method works. Each time we create a new instance of the class, the value returned by the getter increases by one. As shown in Figure 14, after the second instance is created, `getNumberOfInstances`, in *both* instances returns 2.

Let us define another method, `mid`, that finds the midpoint between two existing points (Figure 15 top-left). It takes as parameters the X and Y coordinates of the two points (Figure 15 bottom-left), and returns a new instance of `ACartesianPoint` representing the midpoint (Figure 15 right).

These operations are a bit different from other operations in `ACartesianPoint` such as `getRadius`. What `getRadius` does depends on the particular instance on which it is invoked. That is because it accesses an instance variable; in different instances, the variable may have different values. This is not the case with `mid` or `getNumberOfInstances`, which do not access any instance variables.

Let us use the analogy between Java and physical objects to better understand the difference between the two kinds of operations Figure 16. The `getRadius`, operation is like a "getMileage" operation "invoked" on a car to display the distance traveled by the car. On different cars, this operation returns different values. The `mid` operation is like a "blend" operation that takes two cars and builds a new car whose color is an average of the colors of the two individual cars. We do not invoke this operation on a car; instead we invoke it on the factory for producing cars.
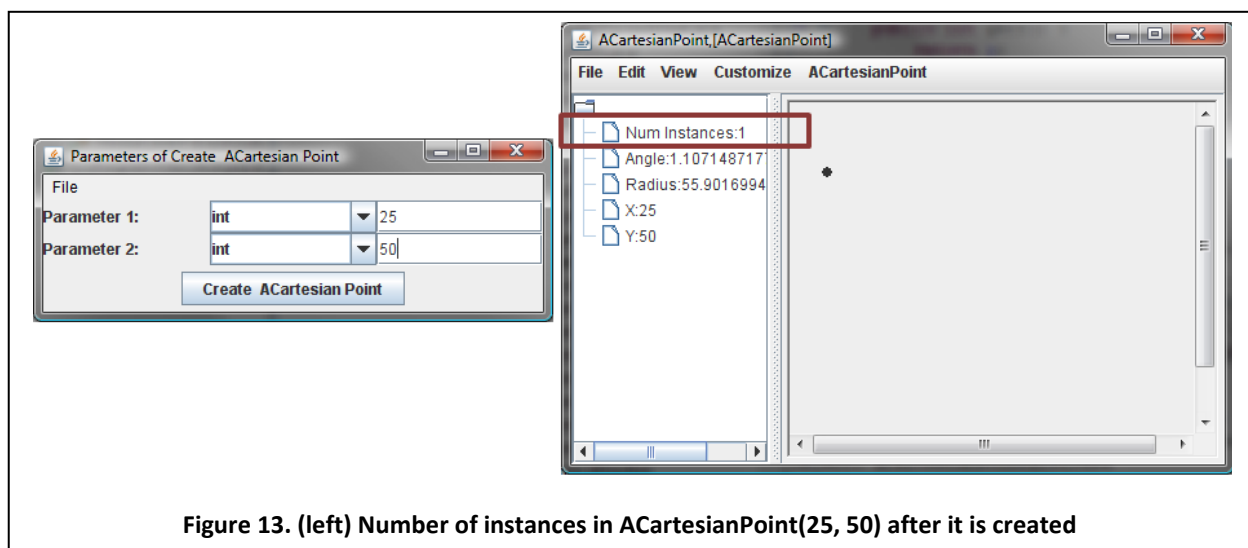

Figure 13. (left) Number of instances in ACartesianPoint(25, 50) after it is created
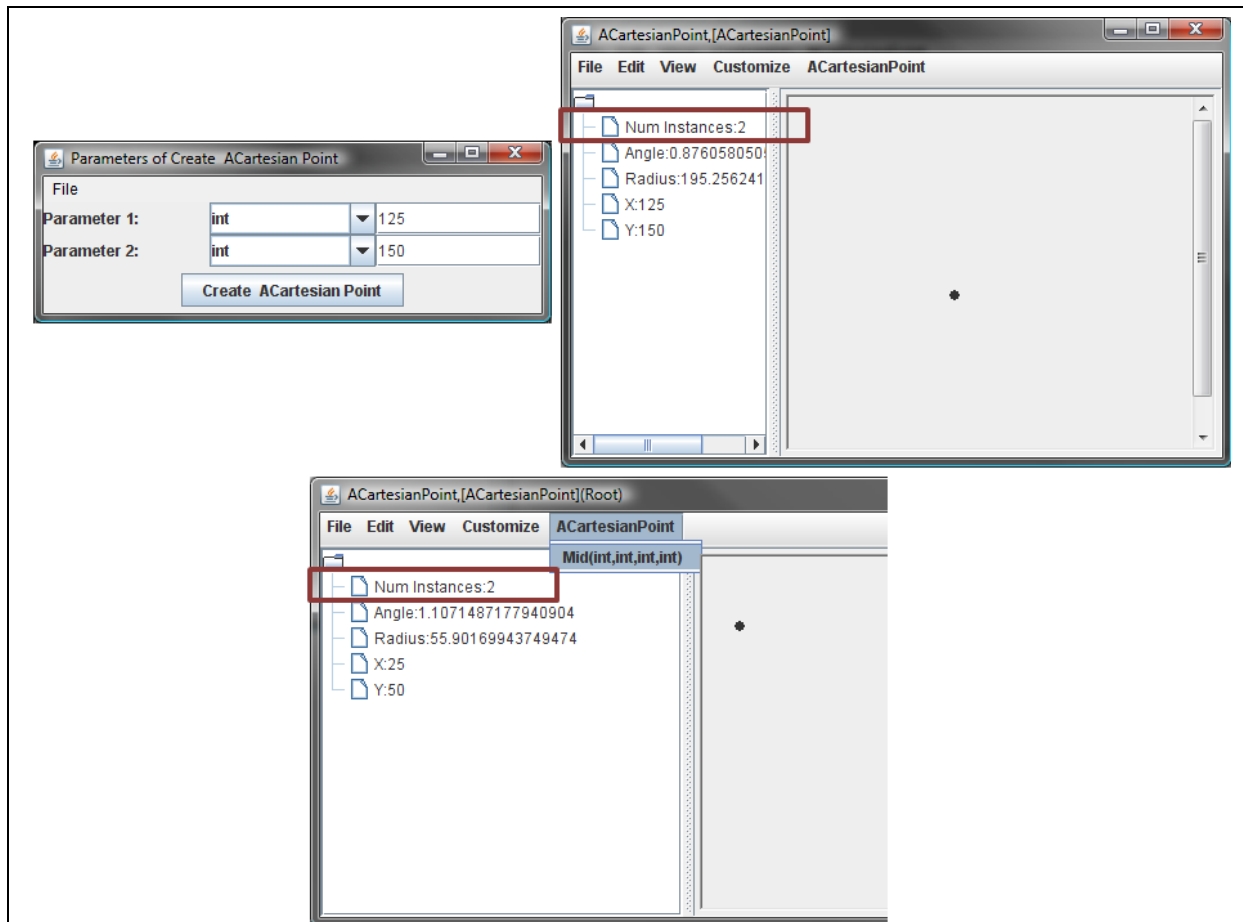
Figure 14. (top) Number of instances in ACartesianPoint(125, 150) after it is created and (bottom) number of instances in ACartesianPoint(25, 50) after a ACartesianPoint(125, 150) is created
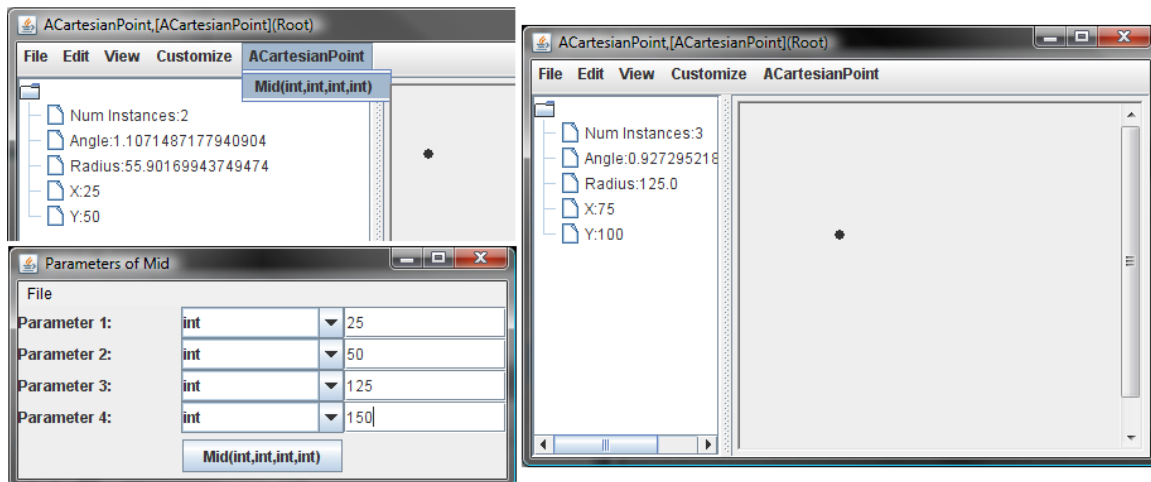

Figure 15. Finding the mid-point between two points: (top-left) calling the mid method

Java similarly, allows us to declare in a class, methods that are not invoked on instances of the class but instead on the class itself (Figure 17). Such methods are called *class methods* or *static methods*. On the
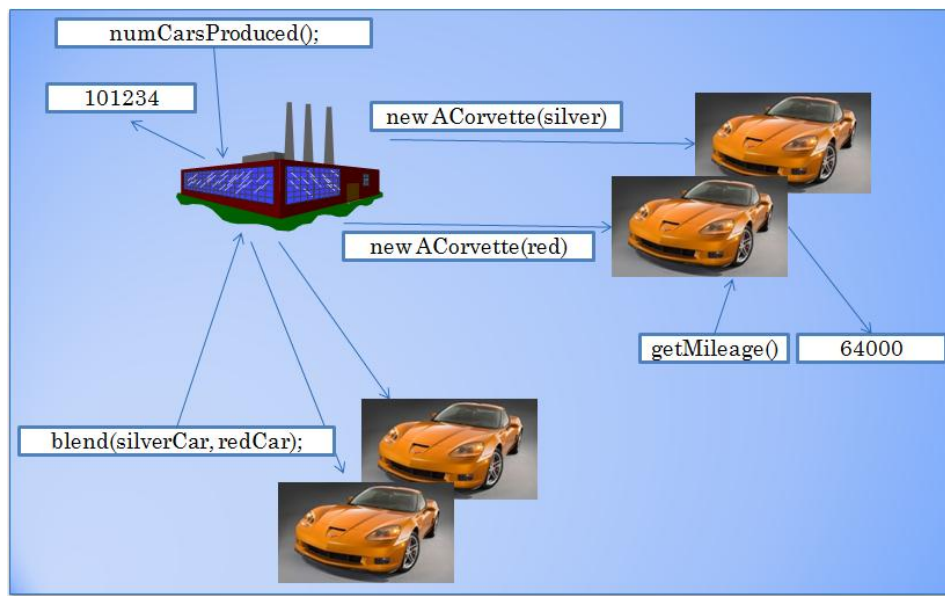
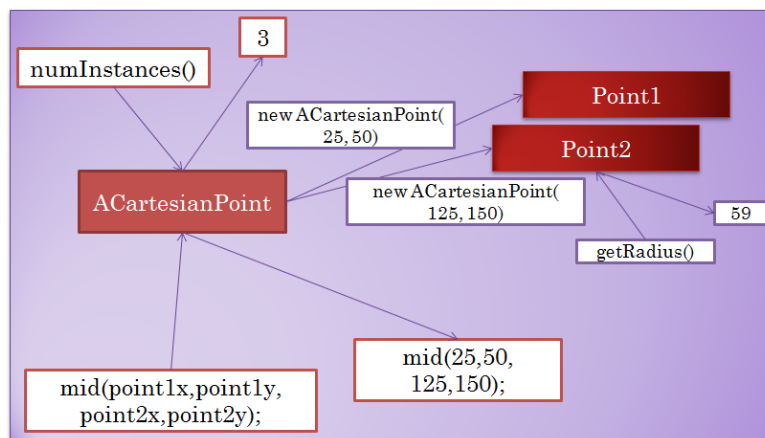**Figure 16. Real-world analogy: factory and object methods**

Figure 17. Java class and instance methods

other hand, methods invoked on instances of the class are called *instance methods*. The definition of a class method is much like the declaration of an instance method except that keyword **static** must be added to the header of the method:

```
public static Point mid (int x1, int y1, int x2, int y2 ) {
    return new ACartesianPoint(x1 + (x2 - x1)/2, y1 + (y2 - y1)/2);
}
```

Unlike instance methods, class methods cannot be listed in an interface.

Once this method is defined in ACartesianPoint, we can invoke it not only interactively, as shown in Figure 15, but also programmatically, as shown below:

```
Point midPoint = ACartesianPoint.mid(25, 50, 125, 150)
```

As illustrated above, the syntax for method invocation is the one we have seen before:

<Target>.<Method>(<Actual Parameters>)

However, this time, the target of the method is a class rather than an instance. We have, in fact, already used this syntax - when we discussed the math library:

Math.round(6.7)

`Math` is a class and `round` a class method invoked on it. Like the add method, what the round method does depends entirely on its argument. Hence it, and all of the other methods in the math library, is a class method instead of an instance method.

Class Variables

Consider now the class method `getNumberOfInstances` that counts how many instances of the class have been created so far. Such a method corresponds to a facility provided by a factory to determine how many cars have been manufactured by it. How might such an operation be implemented in `ACartesianPoint`? The class must keep a count, `numInstances`, of how many instances have been created so far, and update this count whenever a new instance is created. We can imagine defining a special method for updating the count:

```
public static void updateNumberOfInstances () {
        numInstances++;
}
```

However, this approach will work only if we explicitly call this method each time we instantiate the class:

```
new ACartesianPoint(25, 50);
ACartesianPoint.updateNumberOfInstances();
        // must follow each invocation of new
```

However, we might forget, and it is tedious to do so. Moreover, we might accidentally call this method even when no new instance was created.

It is preferable to update the variable in some method that is automatically called when a new class is created – in other words the constructor of the class. Thus, in addition to initializing the instance variables of the new object, the constructor can now maintain the count:

```
public ACartesianPoint(int theX, int theY) {
    x = theX;
    y = theY;
    numInstances++;
}
```

Our implementation of `getNumberOfInstances` is now trivial:

```
public static int getNumberOfInstances () {
    return numInstances;
}
```

It simply returns the current value of `numInstances` – the variable that keeps the count. As its name suggests, this is a getter method; however, unlike other getter methods we have seen so far, it is a class method. We will refer to properties defined by class methods as *class properties* and those defined by instance methods as *instance properties*.

Instance vs. Class Variables

How should `numInstances` be declared in the class? We cannot, of course, make it a local variable of a method, since it is accessed by two methods, the constructor and `getNumberOfInstances`.

We could declare the variable as an instance variable, as we have done with other variables storing values of properties:

```
int numInstances = 0; // instance variable
```

As it turns out, this will also not work in this case. This is because a new copy of an instance variable is created each time the class is instantiated, allowing states of two different instances to diverge. However, here, we want the same variable to be updated on different instantiations of the class.

Fortunately, Java allows us to declare it as a *class variable,* which is created once for the whole class. Like an instance variable, a class variable is declared outside all methods of the class in which it is defined. The difference is that the keyword `static` appears in its declaration.

```
static int numInstances = 0;// instance variable
```

This discussion makes sense intuitively: local variables store the data specific to a particular method call; instance variables store data specific to particular instance but shared by all methods invoked on the instance; and class variables store data shared by the whole class. The number of instances of a class is relevant to the whole class and not a particular method call or instance.

The difference between instance and class state is visually illustrated by Figure 18, which shows edit windows of two different instances of `ACartesianPoint`.

The edit windows show both class and instance properties. As we see, the value of the class property is the same in the two windows, since class state is shared by all instances of the class. On the other hand, the instance properties have different values in the two windows, since instance state is different in the two objects.

Visibility Rules

As we have seen, local variables are visible only in the methods in which they are declared. We had earlier said that instance variables are visible in all methods of the class in which they are declared. That was true as long as we were defining only instance methods. As it turns out, instance variables are not visible in class methods (Figure 19). Moreover, they cannot be used in initializing declarations of class variables. On the other hand, class variables are accessible throughout the class in which they are declared. Thus, they are visible in all methods – both instance and class methods, and can be assigned to instance variables.
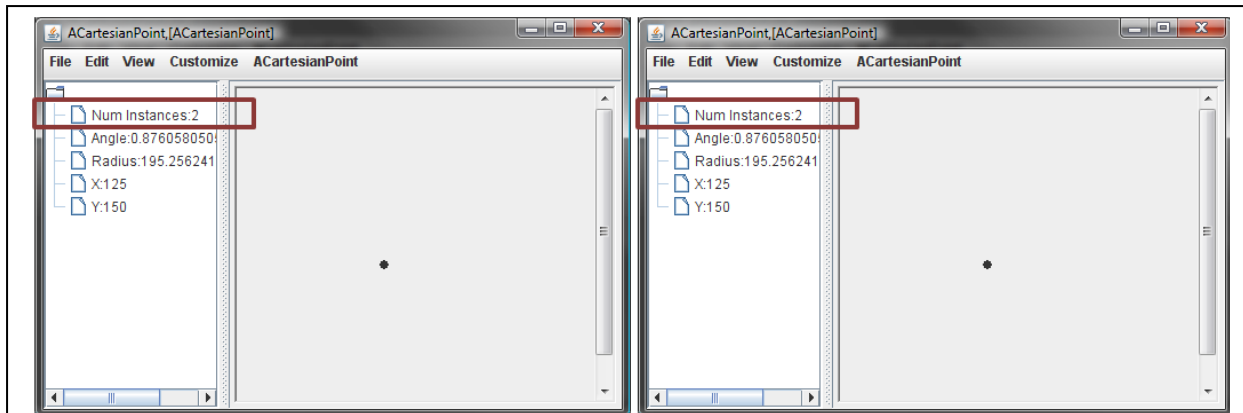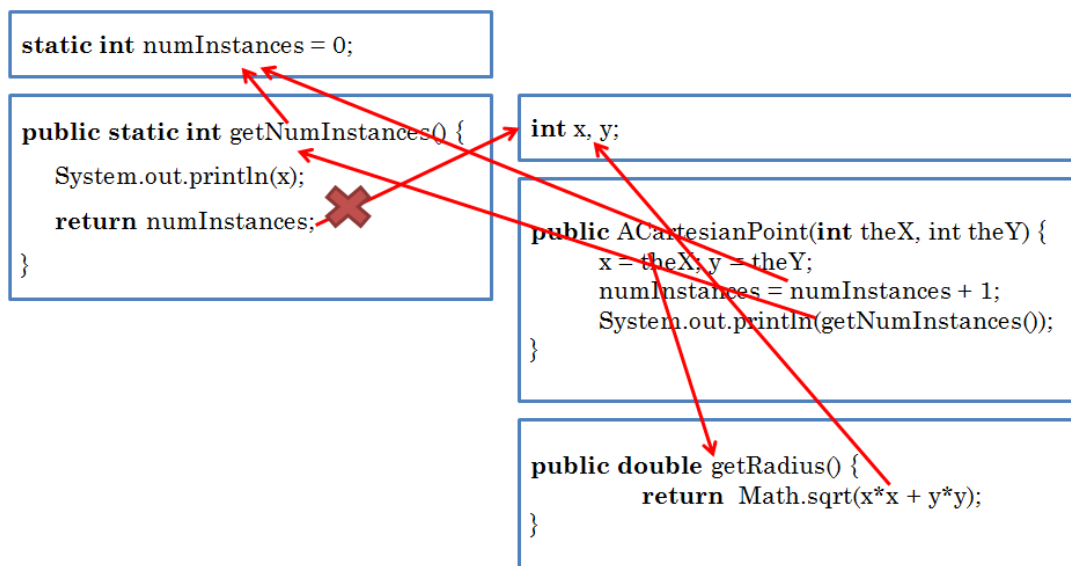
Figure 18. Instance vs. class state

Figure 19. Example of visibility of class and instance members

Why are instance variables also not visible throughout the class in which they are declared? Recall that there are as many copies of an instance variable as there are instances. If an instance variable were accessible in a class method, then which copy of the instance variable should be accessed? With an instance method, there is no such ambiguity, since it accesses the copy of the instance variables created for the instance on which it is invoked. Thus, when we invoke the `setX` method on an instance, it is clear that it should change the X instance variable of that object. Unlike an instance method, a class method is not invoked on a specific instance, but on the class. For similar reasons, a class method can be used in an initializing declaration of an instance variable, but not vice versa, and an instance method can call a class method, but not vice versa. These rules are akin to not allowing an operation invoked on a factory to manipulate parts of cars that have been sold by the factory.

The variables (and named constants) representing the state of an object and the methods that can be invoked on it are called *members* of the entity. Thus, the instance variables and instance methods defined by a class are called members of their instances, while the class variables and methods defined

in a class are called members of the class itself. We can now state our visibility rules in terms of members. A class member is accessible throughout the class in which it is declared. An instance member is accessible only to other instance members. In other words, an instance member can be used in an initializing declaration of only an instance variable and can be referenced only by an instance method. Moreover, an instance method can be called only by another instance method.

Instance vs. Class Methods

Consider the method, `mid` and `getNumberOfInstances`, again. We argued that they should be made class methods because they do not need to access any instance variables. However, because of the visibility rules above, they could have been made instance methods without affecting the behavior of the program. In general, all class methods can be made instance methods because instance methods can access not only instance variables but also class variables. For example, if `getNumberOfInstances` was an instance method, it could still access the variable, `numInstances`.

Thus, we must decide whether all methods of a class should be instance methods, or only those that access instance variables. From a philosophical point of view, the second approach is "cleaner" because it separates the operations that are instance-specific from those that are not. Another way to argue for it is to consider the least privilege principle. By making a method that does not need to access instance variables an instance method, we are giving it more privileges (the privileges to accidentally access instance variables) than it needs. Finally, it is more inconvenient to call an instance method rather than a class method. If `getNumberOfInstances` is declared as a class method, then another class can simply use the class name as a target:

```
ACartesianPoint.getNumberOfInstances()
```

On the other hand, if it is an instance method, the other class must first create an instance of the class in which the method is declared (if such an instance does not exist):

```
(new ACartesianPoint()).getNumberOfInstances()
```

One disadvantage, in Java, of making a method a class method is that it cannot be listed in an interface. Moreover, when we study inheritance, we will see that class methods, unlike instance methods, are not "virtual." If we need to list a method in an interface or make it virtual, then it should be made an instance method, regardless of which variables it accesses.

Instance vs. Class Constants

Similarly, when declaring a named constant, we must decide whether we make it a class member:

```
public static final double LBS_IN_KG = 2.2;
```

or an instance member:

```
public final double LBS_IN_KG = 2.2;
```

When declaring a regular variable, there is no choice – if a separate copy of the variable is needed for each instance, we make it an instance member, and if a single copy is needed, we make it a class

member. However, in the case of a named constant, there is no difference between creating one or multiple copies because all of them have the same value – accessing any one copy is the same as accessing any other copy![2]

There is no good reason for creating multiple copies, which decreases space efficiency. Moreover, if the constant is public, then referencing it as an instance constant requires creating an instance, whereas referencing it as a class constant does not. Therefore a named constant should be created as a class member, unless, as discussed below, you do not wish to violate the least privilege principle by doing so.

Recall that the visibility of a class member is greater than the visibility of an instance member. Thus, we should declare a named constant as an instance member if wish to ensure that no class method can reference it. Similarly, if a named constant must be visible only in one method, we should declare it as a local variable of the method. Because a named constant cannot be changed, it is not so dangerous to violate the least privilege principle when declaring a named constant. Thus, most programmers declare named constants as class members. Whether you wish to restrict access to named constants is up to you – there is no consensus on which way it should be resolved.[3]

Object Parameters and Polymorphism

So far, all the methods we have seen have taken primitive arguments, that is, arguments of **int**, **double,** and other primitive types. The following is an implementation of `mid` that takes object arguments:

```
public static Point mid (Point p1,  Point p2 ) {
        return mid (p1.getX(), p1.getY(),
            p2.getX(), p2.getY());
    }
```

Instead of the X and Y coordinates of the two end points, we pass to this method objects implementing the Point interface. The method extracts the X and Y coordinates of each of its arguments, and calls the previous version of `mid` to compute its return value. Thus, both the arguments and return values of this method are objects.

In this implementation, we followed the principle of using interfaces rather than classes to type formal parameters. This example helps us better understand benefits of this principle. We can pass actual parameters to this method that are instances of both `ACartesianPoint` and `APolarPoint`. Thus, the same method can be used to find the midpoint between two instances of `ACartesianPoint`:

```
Point mid = mid(new ACartesianPoint(25, 50), ACartesianPoint(125, 150));
```

---

[2] Actually Java allows the declaration of an uninitialized instance named constant that must be initialized (once) in the constructor used to create the instance. Such a named constant would indeed have different values in different instances, and would have a different meaning if created as a class constant. The discussion here applies to initialized named constants.

[3] In fact, in an earlier version of Java, one could not create named constants as local variables!

two instances of `APolarPoint`:

```
 Point mid = mid(new APolarPoint(1.1, 55.6), APolarPoint(0.87, 195.25));
```

and an instance of `ACartesianPoint` and an instance of `APolarPoint`:

```
 Point mid = mid(new ACartesianPoint(25, 50), APolarPoint(0.87, 195.25));
```

If we used classes to type these variables, we would need to create three different implementations of this operation, one for each of these three combinations:

```java
public static Point mid (ACartesianPoint p1,  APolarPoint p2 ) {
        return mid (point1.getX(), point1.getY(),
            point2.getX(), point2.getY());
}
public static Point mid (APolarPoint p1,  APolarPoint p2 ) {
        return mid (point1.getX(), point1.getY(),
            point2.getX(), point2.getY());
}
public static Point mid (ACartesianPoint p1,  APolarPoint p2 ) {
        return mid (point1.getX(), point1.getY(),
             point2.getX(), point2.getY());
}
```

Thus, if we use interfaces as the types of variables, we write one piece of code to process instances of multiple classes. Such kind of reuse is called *polymorphism*, and an operation that has one or more formal parameters that can be assigned actual parameters of different types is called a *polymorphic operation*.

Interfaces are only one mechanism for supporting polymorphism. Later, we will see that inheritance offers another such mechanism.

A polymorphic operation should be distinguished from an overloaded operation. In both cases, the operation can be applied to values of different types. The difference is that in the case of a polymorphic operation, one implementation of the operation is created for processing different types of values, while in the case of an overloaded operation, multiple implementations are created. The `mid` method we defined is both overloaded and polymorphic. It is overloaded because it has two implementations, one that takes four **int** parameters, and a second one that takes two Point parameters. The second implementation, in turn, is polymorphic because it takes parameters of type `ACartesianPoint` and `APolarPoint`. It could be replaces with three different overloaded implementations, as we saw above.
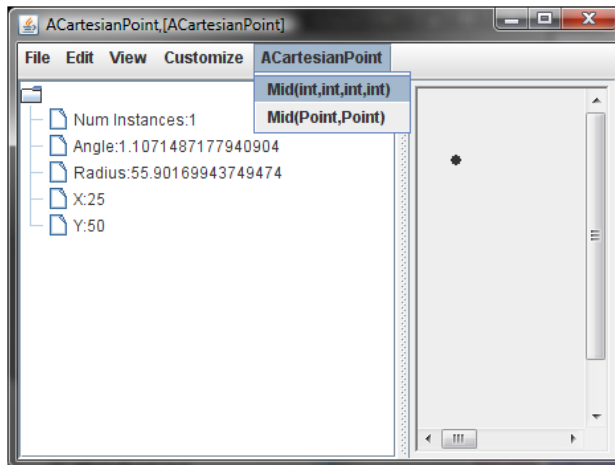
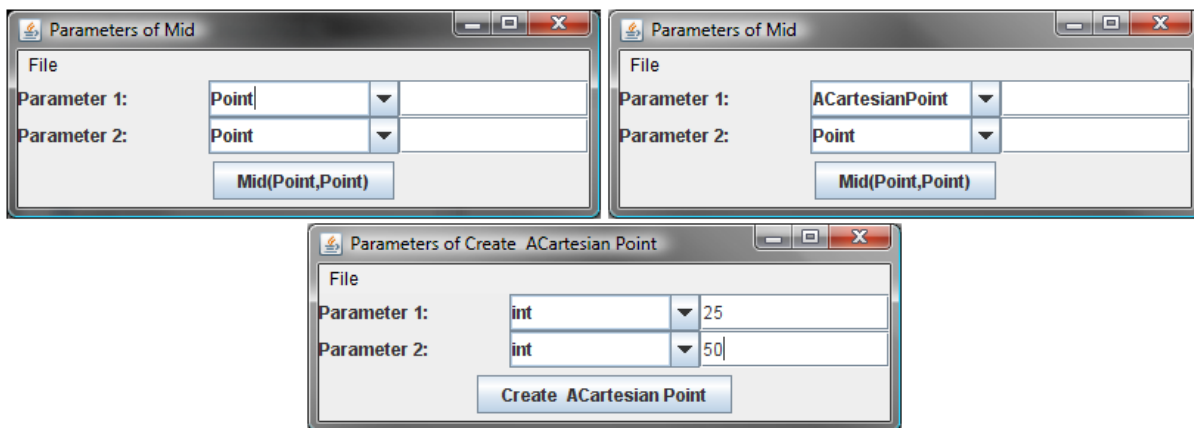**Figure 20. Instance vs. class state**

Figure 21. Example of visibility of class and instance members

Figure 20 through Figure 21 illustrate object parameters and polymorphism graphically. We select the new `mid` from the menu (Figure 20), and as we have seen before, a dialogue box is displayed to enter the actual parameter of the method (Figure 21 top-left). However, this time, the type of the parameter is a polymorphic object type rather than a primitive type. Therefore, before entering the parameter value, the user must first specify the class of the parameter. ObjectEditor allows the user to type the exact class name. Let us specify `ACartesianPoint` as the name of the class of the first parameter (Figure 21 top-right). How should a new instance of this class be specified? It is not possible to simply enter a text string next to the type name as there is no standard string representation of `ACartesianPoint`. The only way to specify such an object is to invoke its constructor. As the class has only one constructor, ObjectEditor automatically chooses it, and asks for its parameters. Thus, at this point, we have two parameter windows: one for the parameters of the original method we invoked, `mid` (Figure 21 top-right), and the other for the parameters of the constructor, `ACartesianPoint` (Figure 21 bottom).

We can now specify the parameters of the constructor it in the manner we saw earlier. When we invoke the constructor to create a new instance of `ACartesianPoint`, ObjectEditor displays the instance and
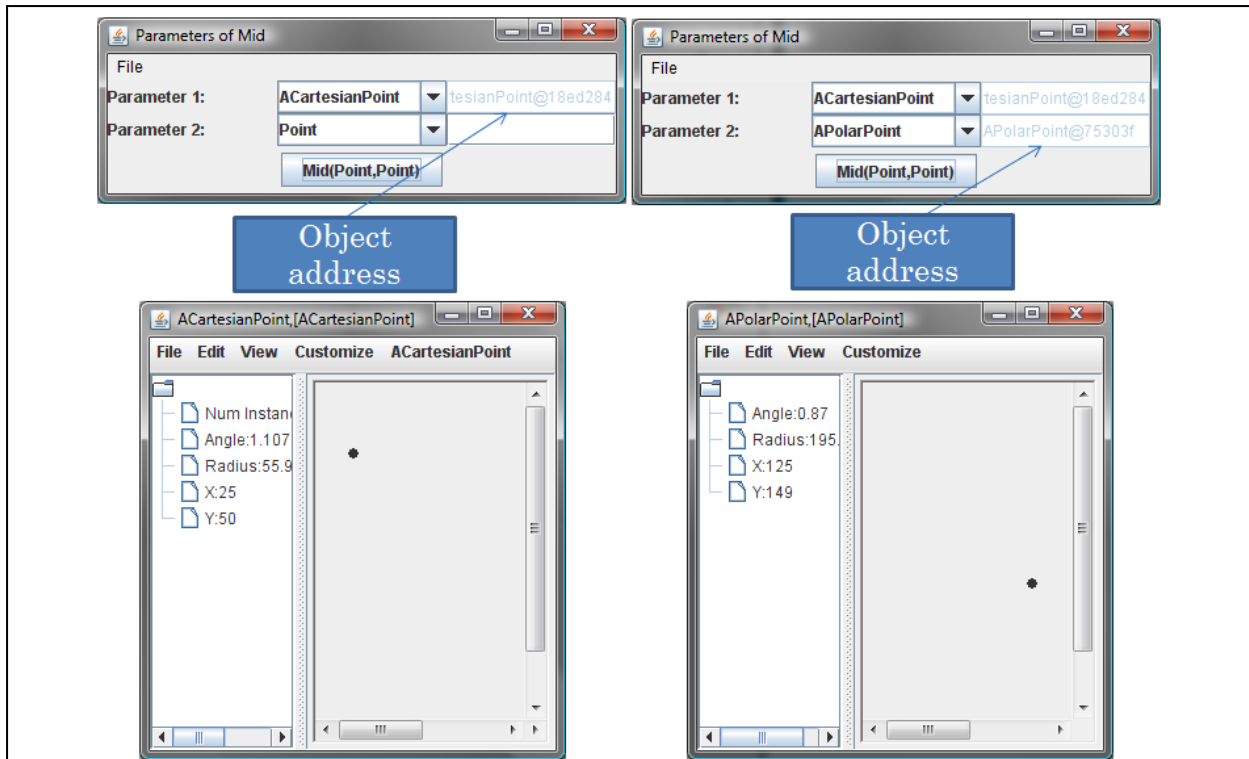
Figure 22. (left) The first parameters of `mid` and the address of the parameter and (right) The second parameter of `mid` and the address of the parameter
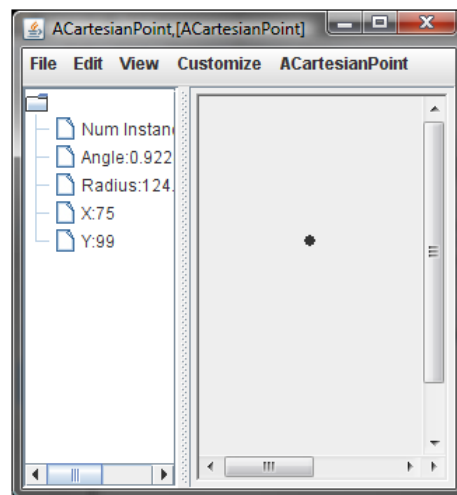

Figure 23. The return value of mid

also enters the address of the instance in the slot for the first parameter of `mid`, as shown in Figure 22 (left). We can similarly enter the second parameter of `mid`, this time choosing `APolarPoint` as the class of the parameter (Figure 22 (right)). As both parameter values of this method have been specified, we can press the button in the dialogue box to call it. As always, ObjectEditor displays the return value of the invoked method (Figure 23).
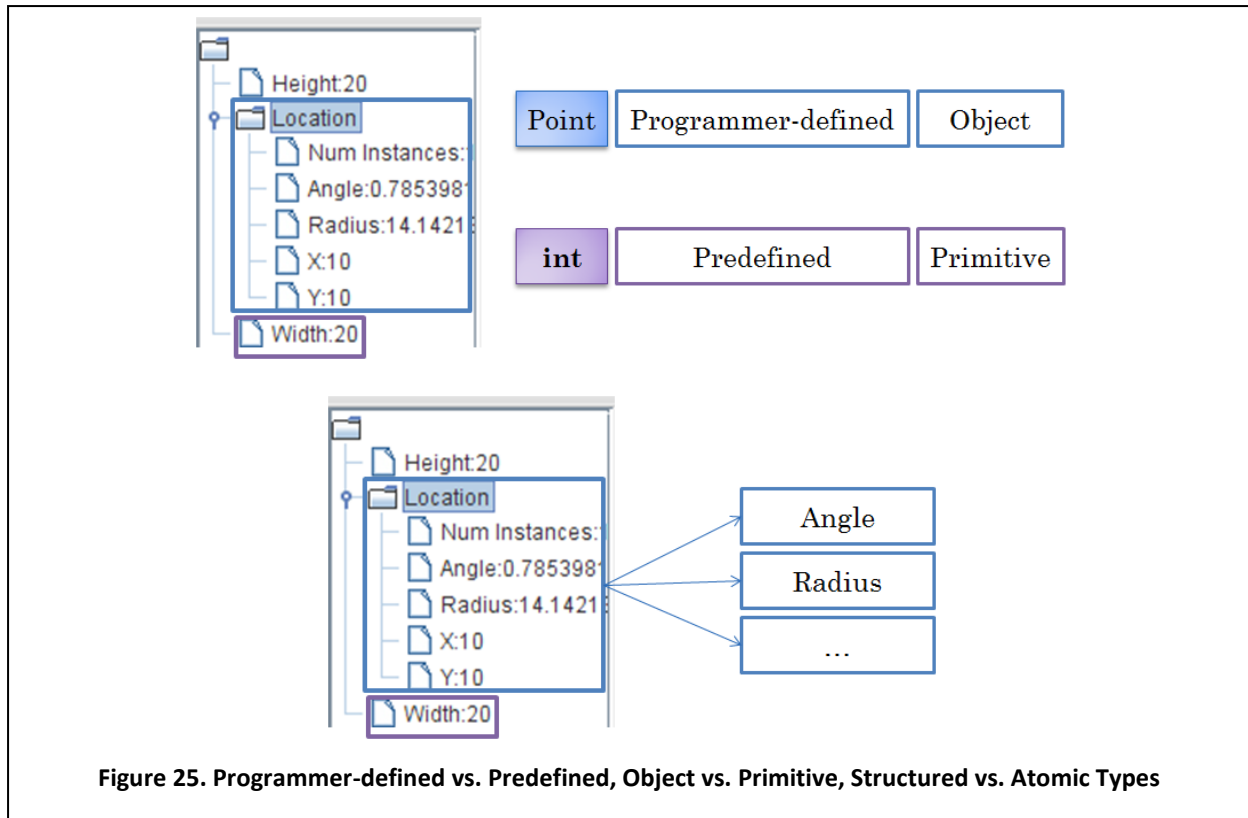
Figure 24. Two representations of a line
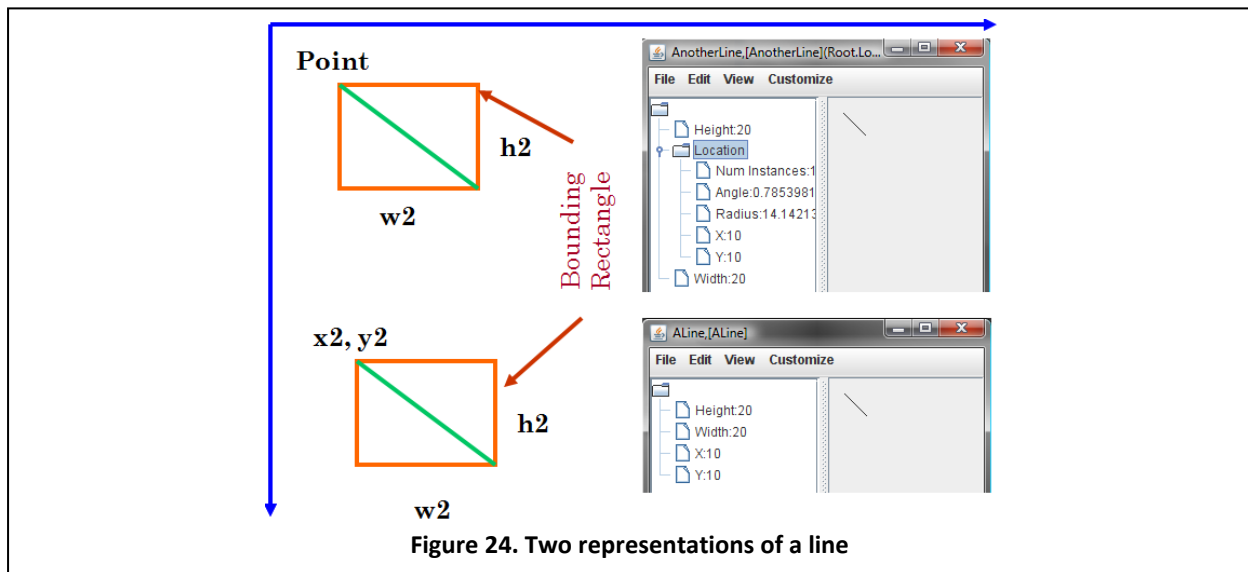
Shapes and Different Kinds of Types

A point is one example of a geometric object. There are many other such objects we may wish to define such as line, rectangle, or oval. Each of these objects has many possible representations. To illustrate, let us take the example of a line. Figure 24 shows two possible representations. Both of them describe the line by its bounding rectangle. In one of them, the upper left corner is represented by an object of type Point while in the other one it is represented by its two integer Cartesian coordinates. In both cases, the width and height of the rectangle are defined by two integer values. The upper left corner of a bounding rectangle is called the location of the rectangle. The pair (width, height) is called the size of the rectangle. If we define properties for the location and size in the two representations, we get the ObjectEditor tree displays shown in the right part of Figure 24.

The following class, `ALine`, uses the representation in which the location of the line is represented by its Cartesian coordinates:

```
public class ALine implements Line {
    int x, y, width, height;
    public ALine (int initX, int initY, int initWidth, int initHeight) {
        x = initX;
        y = initY;
        width = initWidth;
        height = initHeight;
    }
    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 int getWidth() {return width;}
    public void setWidth(int newVal) {width = newVal;}
    public int getHeight() {return height;}
    public void setHeight(int newHeight) {height = newHeight;}
}
```

Figure 25. Programmer-defined vs. Predefined, Object vs. Primitive, Structured vs. Atomic Types

It is very similar to the classes we have seen before, defining getters and setters for the variables representing the object. The class, `AnAnotherLine`, which represents the upper left corner as a Point object, is also very similar:

```
public class AnAnotherLine implements AnotherLine {
    int width, height;
    Point location;
    public AnAnotherLine(
            Point initLocation, int initWidth, int initHeight) {
        location = initLocation;
        width = initWidth;
        height = initHeight;
    }
    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;}
}
```

However, there are two related differences between this class and the ones we have seen before. First, this is the first class in which we have defined an instance variable/property, `location`, that is an instance of an object type (Figure 25 (top)). All other instance variables/properties have been values of
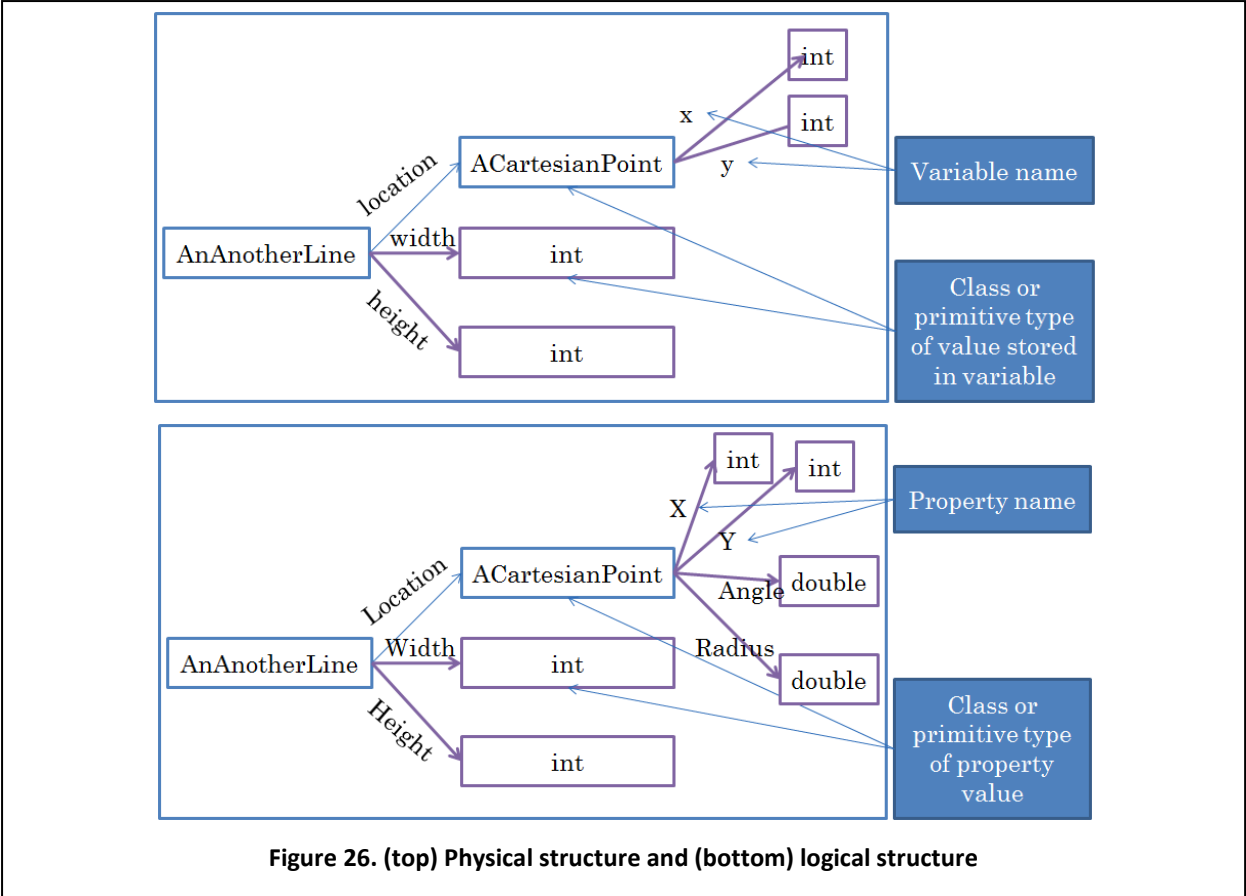
Figure 26. (top) Physical structure and (bottom) logical structure

primitive types. Moreover, the type of the instance variable/property – Point – is a *programmer-defined type*, that is, a type defined by a Java programmer. The types of other instance variables/properties were *predefined types*, that is types defined by Java.

Finally, the instance variable/property can be decomposed into subcomponents such as the Cartesian and Polar coordinates (Figure 25 (bottom)). Thus, its type is a *structured type.* All other instance variables/properties have been indivisible and hence their types have been *atomic types.* In Java, all structured values are objects, and all structured types are object types.

Physical vs. Logical Structure

The notion of structure and atomic types requires more explanation. In ordinary life, we tend to structure objects. A car can be decomposed into wheels, doors, steering wheel, and so on. A wheel, in turn, can be decomposed into a rim, tire, spokes, bolts, and so on. At some point we stop the decomposition, and call the undecomposed objects as atomic objects. For instance, in the car example, a bolt would be considered an atomic object.

Since programming language values often simulate real-word objects, the process of decomposition applies to them also. How do we determine the structure of an object? Consider an instance of the class `AnAnotherLine`. We see in Figure 26 two ways to decompose the object into smaller components.
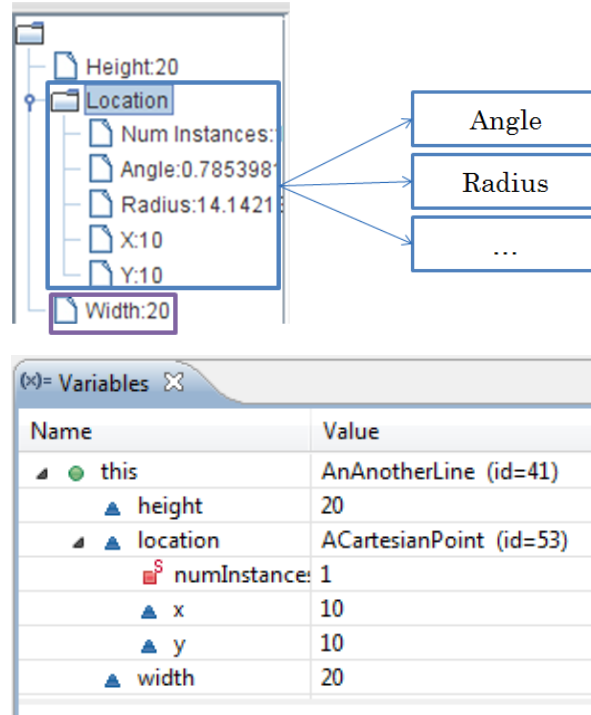
Figure 27. Visualization of logical and physical Structure: (top) logical structure shown by ObjectEditor and (bottom) physical structure shown by debugger
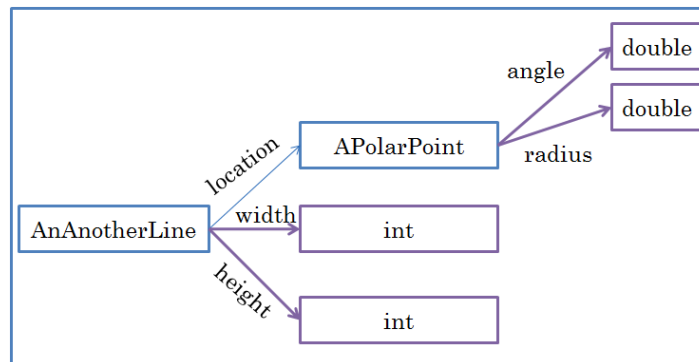

Figure 28. Physical Structure of new **AnAnotherLine (APolarPoint (14.01, 0.78), 20, 20)**

Figure 26 (top) shows the physical structure of the object, which is essentially the structure of its representation in memory. The object is decomposed into the values stored in its instance variables, and each of these values is further decomposed in this fashion, until we reach primitive values. Figure 26 (bottom) shows the logical structure of the object, which decomposes the object into its properties rather than its instance variables. Thus, the object is decomposed into the values stored in its properties, and each of these values is further decomposed in this fashion, until we reach primitive values. Thus, we see that the physical structure of an object depends on its implementation – specifically, its instance variables – while its logical structure depends only on its public methods.

As users of a type, we will be interested in its logical structure, shown by ObjectEditor (Figure 27 (top)), and as its implementers, we will be interested in its physical structure, shown by the debugger (Figure 27 (bottom)). We will use the term component to refer to both a physical instance variable and a logical property and use context to resolve its meaning.

The physical structure of different instances of the same class can be different based on the values of the types assigned to the instance variables defined by the class. For example, the physical structure in Figure 28 is different from the one shown in Figure 26 (top), even though in both cases we are depicting instances of `AnAnotherLine`. The reason is that the location variable has been assigned an instance of `ACartesianPoint` and `APolarPoint`, respectively.

As the figures above show, to draw the physical (logical) structure of a value, we start with the name of the primitive type or class of the value. For each instance variable (property) of the object, we draw an edge from the name. The label of the edge is the name of the instance variable (property) and its end point is the physical (logical) structure of the value of the instance variable (property).

ObjectEditor Geometric-Object Rules

As we have seen above, ObjectEditor recognizes instances of `ALine` and `AnotherLine` as representations of a line. Besides points and lines, it also understands the following geometric objects: rectangles, ovals, text boxes, and labels. These are different from a point in that they are drawn within a programmer-defined bounding box. ObjectEditor assumes that each of these shapes is defined by the width and height of its bounding box and the location of the upper left corner of the box. It assumes that the width and height are represented by **int** properties, named Width and Height (Figure 29). The location can be described either by

- **int** properties, named X and Y, describing its Cartesian coordinates (Figure 29 (top))
- an object property, named Location, whose type describes the point using the ObjectEditor rules for point we saw earlier (Figure 29 (bottom)).

To distinguish between these four geometric objects, it assumes that the name of the interface of class of an object representing a line/rectangle/oval/text-box/label has the string Line/Rectangle/Oval/Text/Label in it. We were very careful in following these rules in `ALine` and `AnAnotherLine` to ensure that instances of these classes are displayed as line shapes. As mentioned before, there are many other useful representations of graphics objects, which ObjectEditor currently does not support.

A textbox has an additional (read-only or editable) String property called, Text, which specifies the string to be drawn inside its bounding box. The text of a textbox can be edited by the user. A label has

- a String property called Text giving the label text, *and/or*
- a String property called, ImageFileName, specifying the name of a graphics file describing the label icon.

These two properties cannot be interactively edited. On the other hand, if these properties are associated with setters, then they can be modified programmatically.

Figure 29. ObjectEditor Representation of Line, Rectangle, Oval and Textbox
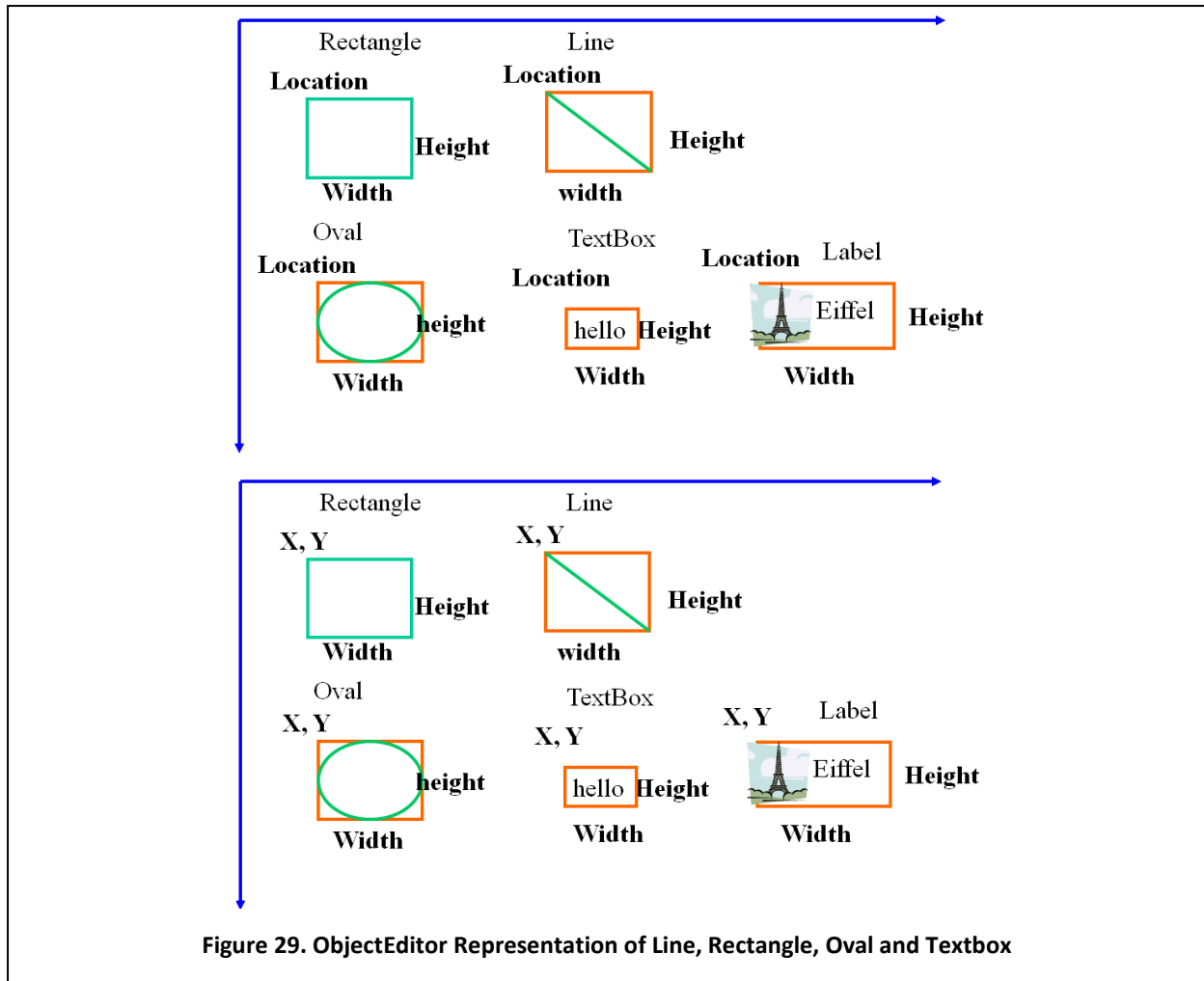
We will see later, in depth, the nature of the type String describing these properties. For now, it is sufficient to know that this type describes text values which are specified in code using a sequence of characters enclosed in quotes, as shown below:

```
String string = "Hello"
```

The following two interfaces follow the label rules.

```
public interface Label {
      public Point getLocation();
      public void setLocation(Point newVal);
      public int getWidth();
      public void setWidth(int newVal);
      public int getHeight() ;
      public void setHeight(int newVal);
      public String getText();
      public void setText(String newVal);
      public String getImageFileName();
      public void setImageFileName(String newVal);
}
```

```
public interface AnotherLabel {
      public Point getLocation();
      public void setLocation(Point newVal);
      public int getWidth();
      public void setWidth(int newVal);
      public int getHeight() ;
      public void setHeight(int newVal);
      public String getImageFileName();
      public void setImageFileName(String newVal);
}
```

The difference between them is that the first interface, Label, defines both the Text and ImageFileName properties while the second one, AnotherLabel, defines only one of them. Our label rules specify that at least one of these properties must be defined.

The following is a straightforward implementation of the Label interface:

```
public class ALabel implements Label {
      int width, height;
      String text, imageFile;
      Point location;
      public ALabel (
            int initX, int initY,
            int initWidth, int initHeight,
            String initText, String theImageFile) {
            location = new ACartesianPoint(initX, initY);
            width = initWidth;
            height = initHeight;
            text = initText;
            imageFile = theImageFile;
      }
      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;}
      public String getText() {return text;}
      public void setText(String newVal) {text = newVal;}
      public String getImageFileName() {return imageFile;}
      public void setImageFileName(String newVal) {imageFile = newVal;}
}
```

The class defines a constructor that takes as arguments specifying the initial values of the properties of a label. Let us invoke this constructor interactively (Figure 30 (left)). Figure 30 (right) shows the display of the created label.

As these figures show, String values specified interactively do not need the quotes around them. ObjectEditor automatically adds the quotes when it creates actual parameters out of them. Moreover,
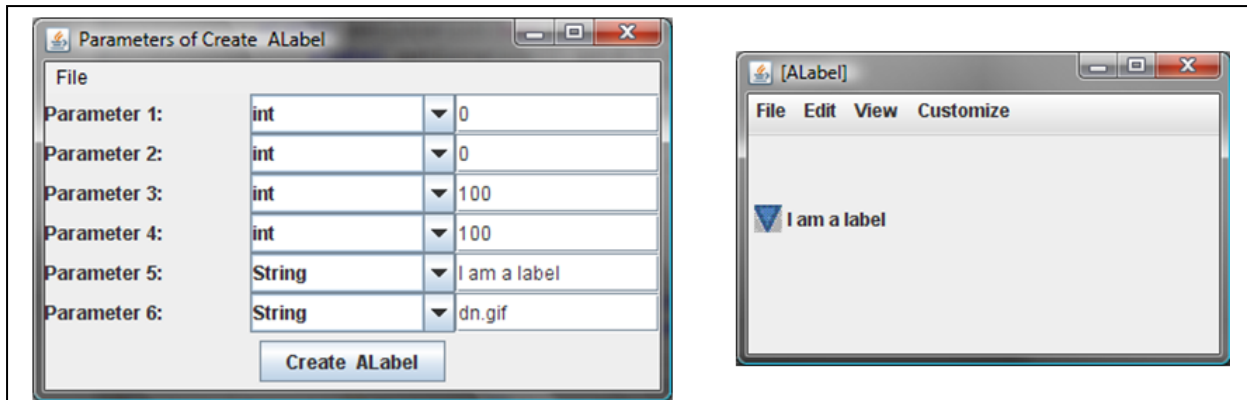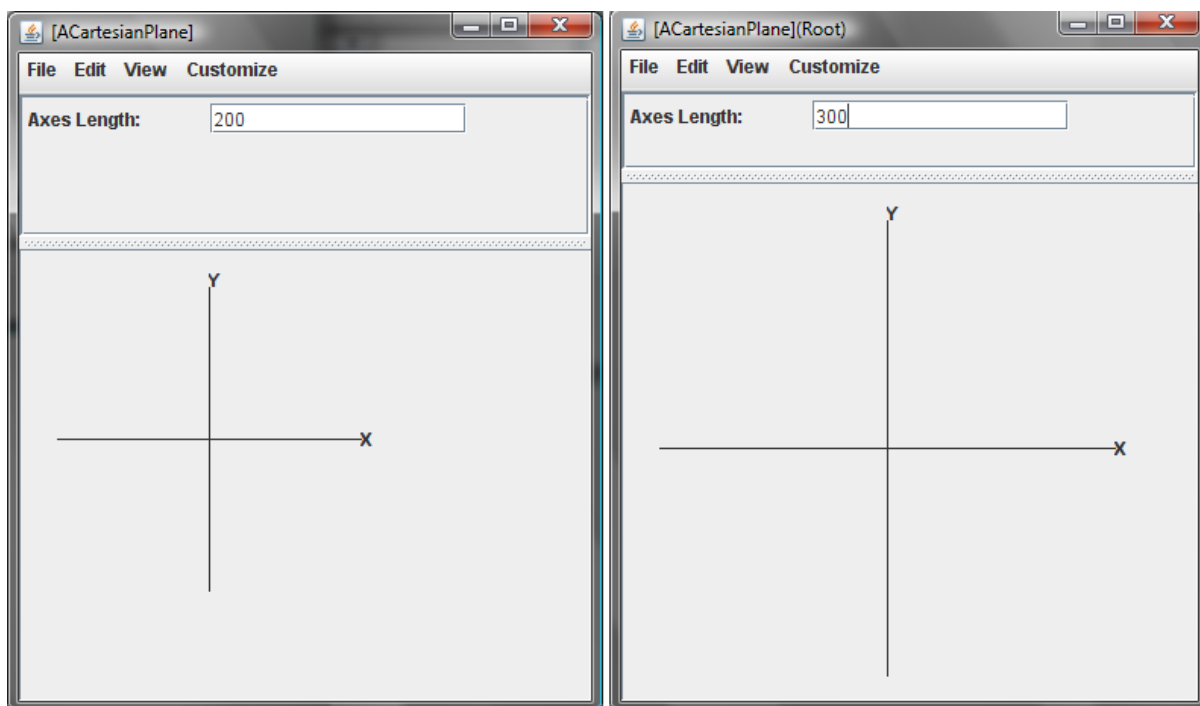
Figure 30. (left) … (right) …

Figure 31. An instance of ACartesianPlane

we also see in these figures how the text and icon of the label are positioned in its bounding box. The X coordinate of the icon is the same as the X coordinate of the upper left corner of the bounding box – in this case, 0. The Y coordinate of the icon, on the other hand, is not the Y coordinate of the upper left corner. It is, in fact, the Y coordinate of the center of the bounding box – in this case, 50. In other words, it is the Y coordinate of the upper left corner plus half the height of the bounding box. The text is positioned immediately on the right of the icon.

Composing Geometric Objects

Given points, lines and other shapes supported by ObjectEditor, it is possible to generate more complex geometries. This is illustrated in the graphics windows of Figure 31 (left) and (right), which shows a labeled Cartesian Plane. The main window of the figures displays a property that determines the length

of the X and Y axes. As the two figures show, changing the value of this property automatically resizes the two axes.The graphics window is defined by four properties, two of which define the lines of the two axes, and two that represent the labels of the axes. We can use the Line and Label interfaces, and `ALine` and `ALabel` classes to implement these properties, as shown below:

```java
public class ACartesianPlane implements CartesianPlane {
    int axesLength ;
    Line xAxis;
    Line yAxis;
    Label xLabel;
    Label yLabel;
    final int LABEL_BOUNDS_SIDE_LENGTH = 10;
    public ACartesianPlane(
        int theAxesLength, int theOriginX, int theOriginY ) {
        axesLength = theAxesLength;
        xAxis = new ALine(
            theOriginX - theAxesLength/2, theOriginY, theAxesLength, 0);
        yAxis = new ALine(
            theOriginX, theOriginY - theAxesLength/2, 0, theAxesLength);
        xLabel = new ALabel(
            theOriginX + theAxesLength/2,
            theOriginY - LABEL_BOUNDS_SIDE_LENGTH/2,
            LABEL_BOUNDS_SIDE_LENGTH, LABEL_BOUNDS_SIDE_LENGTH, "X", null);
        yLabel = new ALabel(
            theOriginX,
            theOriginY - theAxesLength/2 - LABEL_BOUNDS_SIDE_LENGTH,
            LABEL_BOUNDS_SIDE_LENGTH, LABEL_BOUNDS_SIDE_LENGTH, "Y", null);
    }
    public Line getXAxis() {return xAxis;}
    public Line getYAxis() {return yAxis;}
    public int getAxesLength() {return axesLength;}
    public void setAxesLength(int newVal) {
        int lengthIncrease = newVal- axesLength;
        xAxis.setX(xAxis.getX() - lengthIncrease/2);
        yAxis.setY(yAxis.getY() - lengthIncrease/2);
        xLabel.setLocation(new ACartesianPoint(
            xLabel.getLocation().getX() + lengthIncrease/2,
            xLabel.getLocation().getY()));
        yLabel.setLocation(new ACartesianPoint(
            yLabel.getLocation().getX(),
            yLabel.getLocation().getY() - lengthIncrease/2));
        axesLength = newVal;
        xAxis.setWidth(axesLength);
        yAxis.setHeight(axesLength);
    }
    public Label getXLabel() {return xLabel;}
    public Label getYLabel() {return yLabel;}
}
```
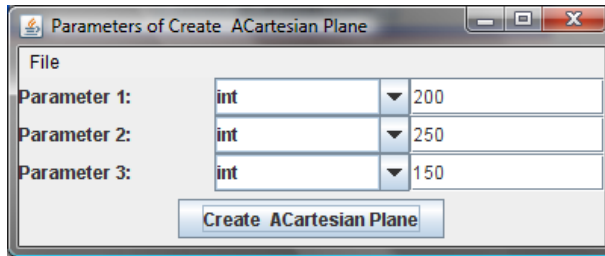
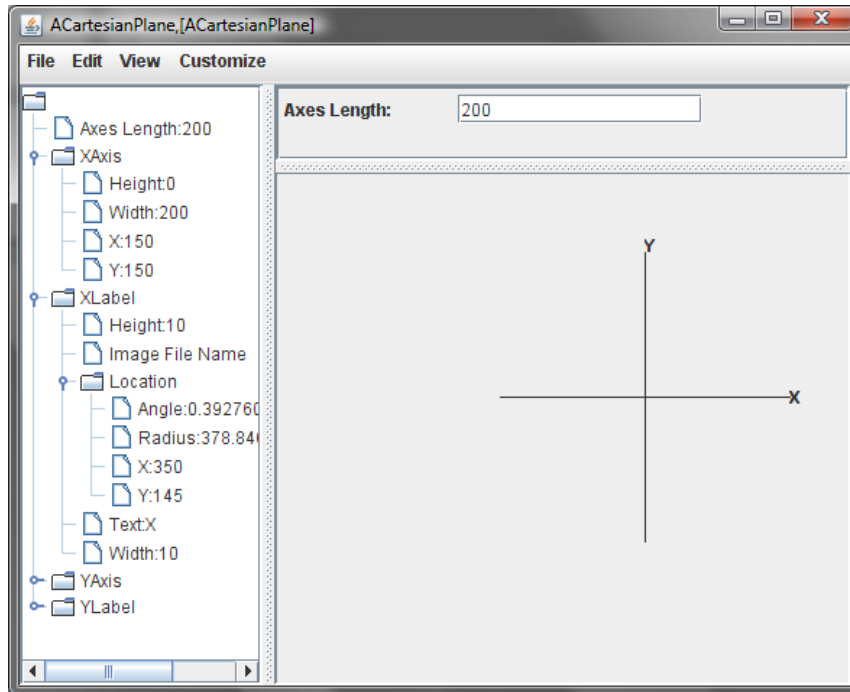**Figure 32. Parameters of the constructor for ACartersianPlane**

Figure 33. An instance of ACartesianPlane

The class declares an instance variable for each of the five properties. It also defines a constant representing the length of the side of the square bounding box of each of the two labels.

Its constructor takes as parameters the initial length of the two axes and the fixed location of the origin of the Cartesian plane. Based on these values, the constructor creates the two lines and labels, and assigns them to the appropriate variables. As the two labels do not display icons, it passes the null value for the name of the image file name.

Figure 32 shows the use of the constructor to create an instance of `ACartesianPlane`. The constructor specifies the initial axes length to be 200 and the origin to be (250, 150). As a result, the width of the XAxis is 200, and the upper left corner, or essentially left corner, of its bounding 0-height box is (250 – 100 , 150); and the height of the YAxis is 200 and the upper left corner, or essentially upper corner, of its bounding box is (250, 150 – 100). As we see in Figure 33, the tree view shows the precise values of the components of all of the graphical properties in a textual display of these properties.
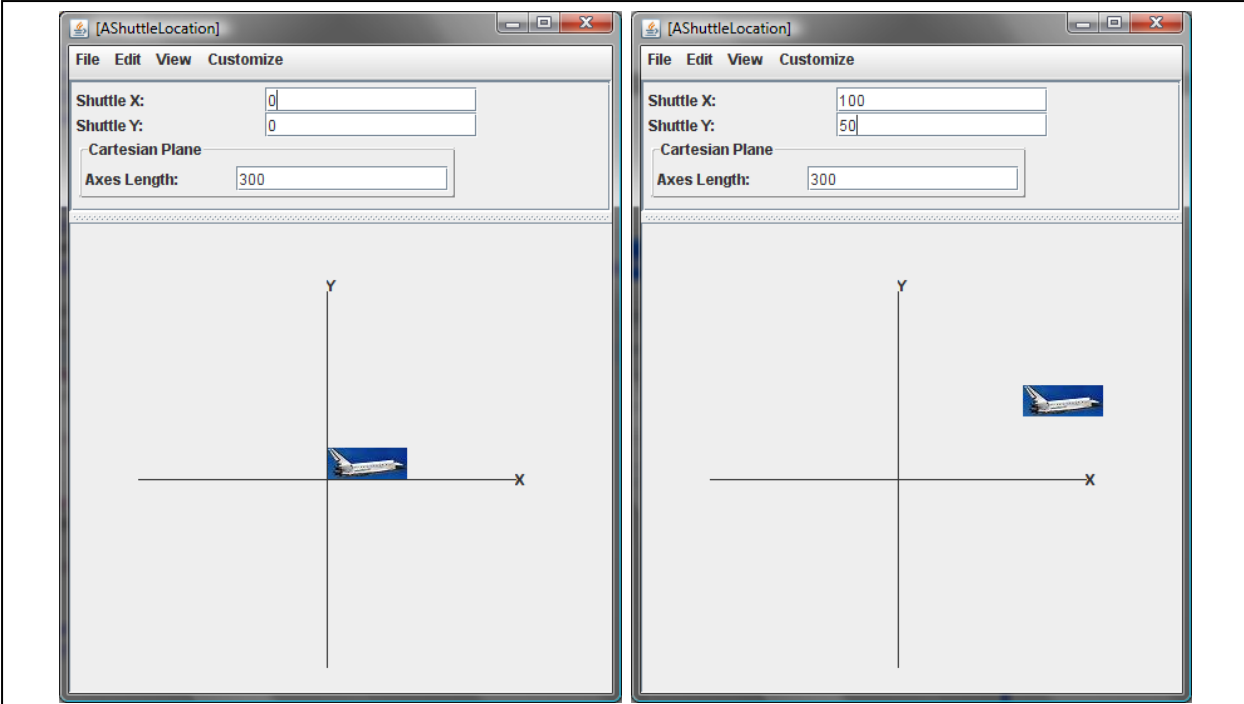
Figure 34. An instance of AShuttleLocation shown in ObjectEditor: (left) Initial shuttle location; (right) shuttle is at location specified by Cartesian coordinates (100,50)
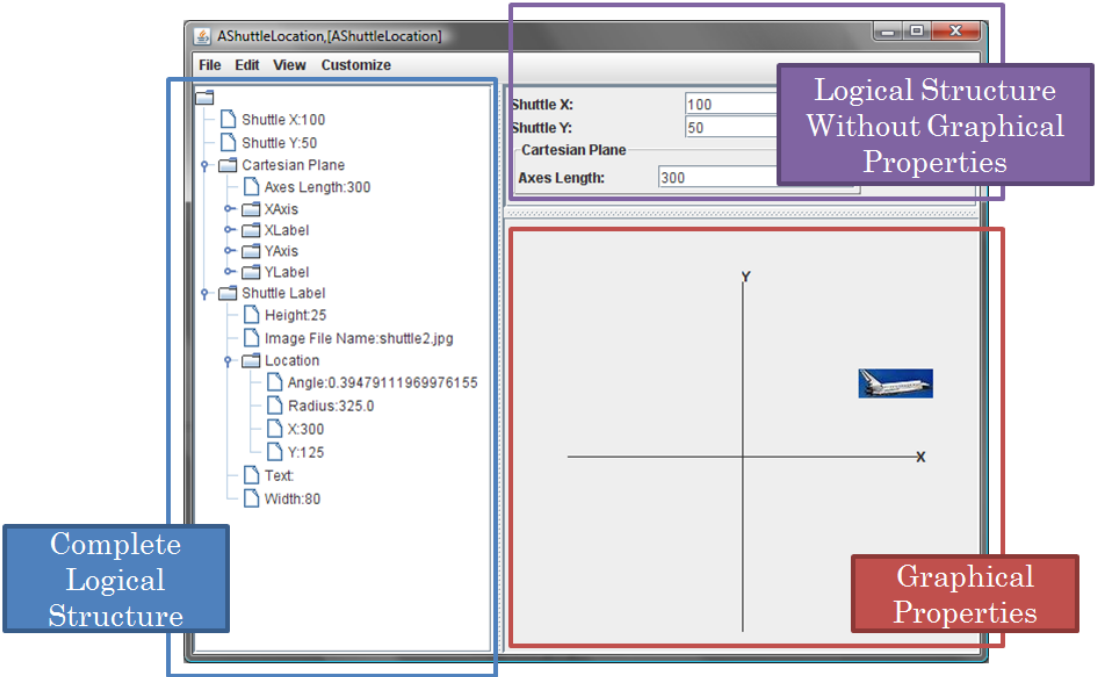

Figure 35. Alternate Views of Logical Structure

Just as we have created `ACartesianPlane` from existing geometric objects, it is possible to compose a larger graphical object from `ACartesianPlane`. Figure 34 shows such an object. It shows the position of a label representation of a (space) shuttle in a 2-D Cartesian plane. The object defines properties, ShuttleX, and ShuttleY, to represent the Cartesian coordinates of the shuttle. Editing them repositions the shuttle label, as shown in the figure.

We can reuse the interfaces/classes, `Label`, `ALabel`, `CartesianPlane`, and `ACartesianPlane`, to implement the class of the above object. Specifically, we can implement a property of type `CartesianPlane`, to define the Cartesian space, and a property of type `Label`, to define the shuttle label. The class, thus, defines four properties in all, shown in the tree view of Figure 35.

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 "Cartesian" space. Thus, the (100, 100) location for the shuttle represents a distance of 100 (pixels) above and to the right of the origin of the Cartesian Plane drawn in the graphics window and not a distance of 100 below and to the right of the upper left corner of the graphics window. In fact, because the Java Coordinates of the origin of the CartesianPlane are (200, 200), the Java coordinates of the shuttle label are (200 + 100, 200-100), as shown in the display of the location property of the label in the tree view. Thus, our new class must translate between the two coordinate systems when repositioning the shuttle label. The code of this class is given below:

```java
public class AShuttleLocation implements ShuttleLocation {
    static final int ORIGIN_X = 200, ORIGIN_Y = 200;
    static final int AXES_LENGTH = 300;
    static final String SHUTTLE_IMAGE = "shuttle2.jpg";
    static final String SHUTTLE_TEXT = null;
    static final int SHUTTLE_WIDTH = 80, SHUTTLE_HEIGHT = 25;
    int shuttleX = 0; int shuttleY = 0;
    ACartesianPlane cartesianPlane = new ACartesianPlane (
        AXES_LENGTH, ORIGIN_X, ORIGIN_Y);
    ALabel shuttleLabel = new ALabel(
        labelX(), labelY(),
        SHUTTLE_WIDTH, SHUTTLE_HEIGHT, SHUTTLE_TEXT, SHUTTLE_IMAGE);
    public ACartesianPlane getCartesianPlane() {return cartesianPlane;}
    public ALabel getShuttleLabel() {return shuttleLabel;}
    public int getShuttleX() {return shuttleX; }
    public void setShuttleX(int newVal) {
        shuttleX = newVal;
        shuttleLabel.getLocation().setX(labelX()); }
    public int getShuttleY() {return shuttleY;}
    public void setShuttleY(int newVal) {shuttleY = newVal;
        shuttleLabel.getLocation().setY(labelY()); }
    int labelX() {return ORIGIN_X + shuttleX;}
    int labelY() {return ORIGIN_Y - shuttle - SHUTTLE_HEIGHT;}
}
```

Figure 36. Composing triangles to create a complex shape

The class defines the expected instance variables and getters/setters for the four properties. The constructor initializes the two object instance variables by creating appropriate instances of `ACartesianPlane` and `ALabel`. 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, and the text, icon and size of the bounding box of the shuttle label, The function `labelX` (`labelY`) translates between the mathematical X (Y) coordinates and Java X and Y coordinates. It is called by both the constructor and setter for ShuttleX (ShuttleY) to ensure that the location of the shuttle label is consistent with value of ShuttleX (ShuttleY).

Figure 35 illustrates the general rule ObjectEditor uses to displays an object's logical structure. It textually displays the complete logical structure of the object in the tree window. It shows all graphical properties in the structure, that is, properties whose types follow some ObjectEditor rule for a geometric object, in the drawing window. In this example, these includes the XAxes, XLabel, YAxes, and YLabel (sub)properties of the CartesianPlane property. It displays the logical structure without the graphics properties in the main window. In this example, it includes the ShuttleX and ShuttleY properties and the AxesLength subproperty of the CartesianPlane property.

As we have seen in this section, it is possible to create complex geometric objects such as `ACartesianPlane` and `AShuttleLocation` in the graphics window from smaller ones such as `ALine` and `ALabel`. In fact, we can create arbitrary computer graphics from the objects we have defined above. 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 36 gives the intuition behind this idea.

Exercises

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

2. Consider the following class:

```java
public class StaticPractice {
    static int s = 0, n = 0;
    int a = 0;
    public StaticPractice (int v) {
        s = s + v;
        n = n + 1;
        a = s/n;
    }
    public static int getS() {
        return s;
    }
    public static int getN() {
        return n;
    }
    public int getA1() {
        return a;
    }
    public int getA2() {
        return s/n;
    }

}
```

Classify the methods of this class into constructors, instance methods, and class methods. Also classify the variables of the class into instance and class variables.

Suppose we create three instances of this class, passing to the constructors the values 10, 20, and 30, respectively. What will be the value of the four properties in each of these three instances? Use ObjectEditor to actually create the three instances. Refresh each edit window after all three windows have been created. Now verify if your calculations were right. Explain the values of the properties for each of the instances. (The names of the variables are deliberately obscure so that you trace the program rather than guess the results based on the names.)

3. Which of the methods of the various classes we saw in Chapter 2: `ABMICalculator`, `ABMISpreadsheet`, etc, could have been class methods?

4. 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();
    }
}
```

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

6. 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.

7. 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.

8. 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.

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`).

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

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