Chapter 2 User Interfaces and Object Editor

For many of us, a computer program becomes interesting when it is interactive. In an interactive program, we must worry about the user-interface. This interface is different from the concept of a programming interface we saw in the previous chapter. These two interfaces, in fact, are related in that a user-interface can be generated from a (programming) interfaces. We will see here a tool, called ObjectEditor, that supports such generation. This tool will be a fundamental instrument for creating interesting non-trivial user-interfaces automatically, checking that interfaces follow style rules, and demonstrating the usefulness of separating user-interface code from other code. Unlike the editors you may have used so far, such as Windows Notepad or Microsoft Word, ObjectEditor is not an editor of text. Thus, it is not meant for changing the code we write to create a class. Instead, it allows us to create and manipulate instances of a class.

Programming Interface vs. User-Interface

The different between user-interfaces and programming is that they are the interfaces through which the user rather than the programmer manipulates objects. The design and implementation of user-interfaces is a difficult and tedious problem. To reduce this problem, interactive systems provide I/O libraries to create console-based user interfaces, and user-interface toolkits to create graphical user-interfaces. However, surveys have shown that, on average, about fifty percent of programs that user the I/O libraries/toolkits is devoted to handling the user-interface tasks such as creating a display, mapping input commands to application functions, and displaying results of these commands. This problem is aggravated by the fact that user-interfaces of applications keep changing.

Designers of programming courses must decide how class and homework time is divided between addressing fundamental programming concepts and creating interesting and meaningful interactive applications motivating these concepts. This decision would not be necessary if a tool could generate user-interfaces. ObjectEditor is such a tool, developed at UNC by the author. It can generate user-interfaces completely automatically or semi-automatically. In this material, we will focus mainly on automatic generation, as the use of the ObjectEditor is not what we are studying. However, some aspects of semi-automatic generation will be described for those who wish to customize their user-interfaces. The use of these features will not be rewarded through regular or extra-credit points to drive home the fact that this course is about programming concepts, and ObjectEditor is simply a tool for learning and motivating these concepts.

Methods to Menus

The idea of automatic generation seems like magic, but actually several aspects of code map to user-interface elements. In particular, a method declaration maps to a menu item – a method name maps to a menu command and parameters of the method map to a dialogue box invoked to gather arguments of a command.

To illustrate, consider the following class, which defines an instance method for calculating a BMI, and a main method to test the calculation method.

The most interesting part of the code is the call:

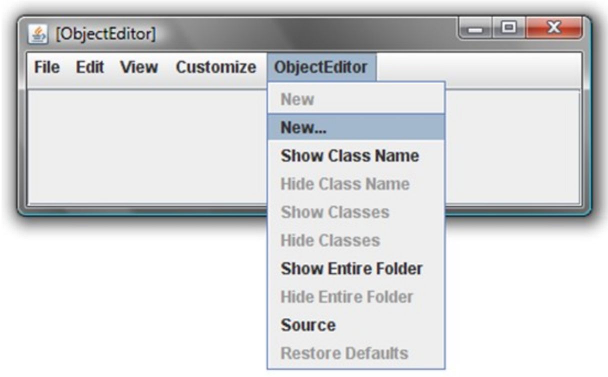
ObjectEditor.edit (**new** ABMICalculator());

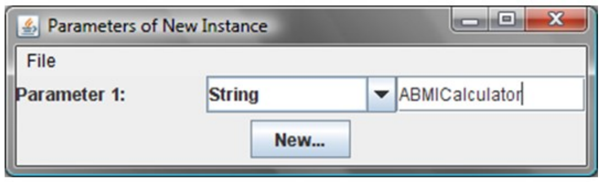
It invokes the static edit method in the imported class, ObjectEditor. This call creates a user-interface to interact with the object passed as a parameter. This user-interface consists of a new window, shown in Figure ???, to represent the object, which we will refer to as the *edit window of the object*.

Figure 2-1. (left) Specifying the parameter and (right) viewing the result

The window provides the menu, ABMICalculator, which consists of a single menu item, Calculate BMI (double, double), which is used to invoke the method of the corresponding name provided by the class. The text in braces, (double, double), after the command name indicates that the method takes two decimal parameter. When we select the operation from the menu, a new window, shown in Figure ??? (left), is created, which prompts the user for the values of the two actual parameter values. Next to each parameter name is a combo-box, which may lead you to believe that you have choice regarding what type of parameter value is actually input. This is not the case for int, double and other primitive parameters, as only one type of actual parameter can be assigned to primitive formal parameter. In the case of a formal parameter typed using an interface, however, instances of different classes implementing the interface can be assigned. For such a parameter, a combo-box can show multiple alternatives.

Let us enter 74.98 (Kg) as the first parameter, weight, and 1.94 (meters) as the second parameter, height (Figure 2-3 left). Clicking on the Calculate BMI button executes the function and displays the value returned by it (Figure 2-3 right).





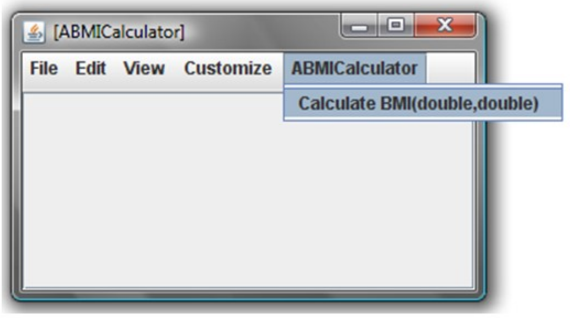
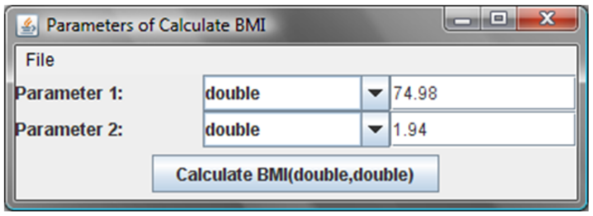
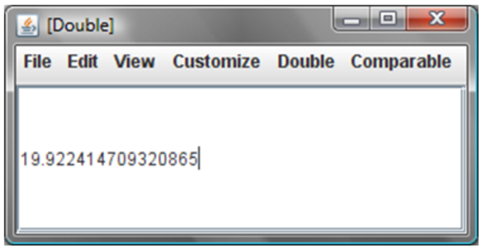


Figure 2-2. Instantiating ABMICalculator



Parameter 1: double 74.98
Parameter 2: double 1.94



```
19.922414709320865
```

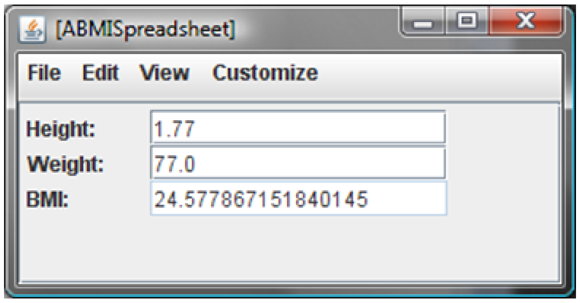
Figure 2-3. Entering actual parameters and invoking calculateBMI

The term, "editor," in ObjectEditor, might seem like a bit of a misnomer, at this point. The reason is that our object had no state to edit. So far, ObjectEditor has simply behaved as an object "driver" or "tester" allowing us to invoke methods in it repeatedly without having to repeatedly change and execute the a driving/testing main method. Below we see how it can be used to edit the state of an object.

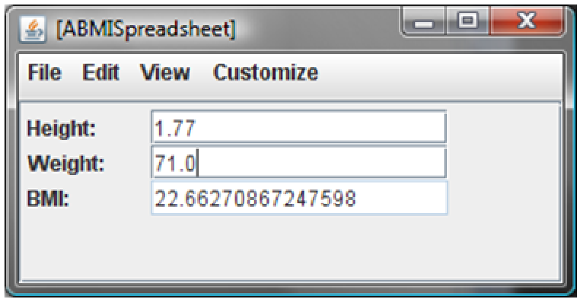
Object Properties to Editable Spreadsheet/Form Items

As we saw before, a class defines both a physical state, consisting of typed instance variables, and logical state, consisting of typed properties.; and the physical state is not exposed to other classes so that it can be changed without affecting these classes. It makes even less sense, then, to expose the physical state to end-users. Thus, ObjectEditor displays the logical properties rather than instance variables of an object. It creates and displays a form item (strictly speaking a "widget" in GUI-toolkit parlance) in the edit window for each property, and allows each editable property to be changed by the user. Each time the user changes an independent property, ObjectEditor displays the values of the dependent properties.

This interaction is illustrated using an instance of ABMISpreadsheet. Recall that it defines three properties, Weight, Height and BMI, with the first two being editable independent properties and the last being a readonly dependent property. Figure ??? shows the user-interface ObjectEditor creates for an instance of this class.



Height: 1.77
Weight: 77.0
BMI: 24.577867151840145



Height: 1.77
Weight: 71.0
BMI: 22.66270867247598

Here, the three properties are shown as form items in an ObjectEditor window. To try a different weight, all we have to do is edit the old value, and press Enter. The object uses the new weight and the current height to re-compute the value of the BMI form item, which ObjectEditor displays. We can try other values of the weight in the same fashion.

Editable properties can be changed by users while those for read-only properties cannot. Thus, in Figure ???, the text field for BMI cannot be changed by users. When we edit a text field of an editable property, a "*" (asterisk) appears next to the name of the property to indicate that field has been changed. When we next hit Enter, the following sequence of actions takes place:

1. The getter method of the property is called first! This may seem strange because, after all, we are trying to change the current value of the property, not read it. The reason has to do with ObjectEditor's undo-redo mechanisms. In order to support undo-redo, ObjectEditor calls the property's getter method first to save the old value.
2. The setter method of the property is called with the value of the text field as the actual parameter of the method.
3. The getter methods of all properties are called and their return values used to refresh the text fields displaying their values.
4. The * next to the name of the changed property disappears to indicate that the display of the property is consistent with its actual value.

We can now understand what exactly happens when we change weight property from 77.0 to 71.0 and hit Enter (Figure ??? (top-left and top-right)).

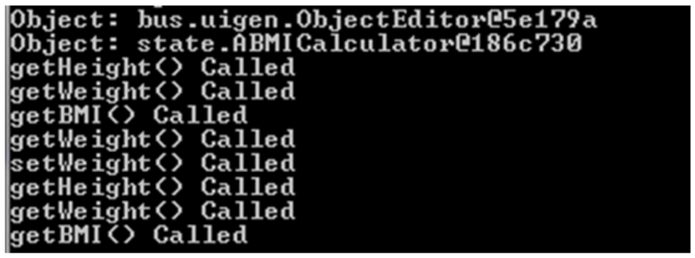
1. The getter method, `getWeight()`, is called first by ObjectEditor's undo-redo mechanism, recording the old value of 77.0.
2. The setter method, `setWeight()` is called, with the parameter, 71.0. The method assigns this value to the instance variable, `weight`.
3. All of the getter methods, `getWeight`, `getHeight`, and `getBMI` are called, and their results are used to refresh the text fields displaying weight, height, and BMI, respectively. There is no change in the first two text fields, since they were displaying the current values of their properties. The text field of BMI, though, does change, because a new value of `weight` is used in the calculation **Error! Reference source not found.** (bottom). Ideally, ObjectEditor should have called only `getBMI`, since BMI is the only property that needs refreshing. However, it does not know the dependencies among the various properties of an object, and assumes that when one of them changes, all of them may change, and thus need to be refreshed.

The getter methods are also called when the object is first displayed in an edit window and when the refresh command is executed.

We can, in fact, trace the sequence of invocations of methods called by ObjectEditor by putting special statements in the getters and setters. For instance, we can insert such a statement in the beginning of the method `setWeight()`:

```
public void setWeight(double newVal) {
        System.out.println("setWeight called");
        weight = newVal;
}
```

If we put similar statements in all of the methods of the class, then we can trace the execution of the methods in the command interpreter window. Figure ??? shows the initial calls to the getter methods when the edit window of `ABMISpreadsheet` instance is first created and the calls made when the weight field is changed. As we can see, this trace is consistent with the sequence of actions mentioned above.



```
Object: bus.uigen.ObjectEditor@5e179a
Object: state.ABMICalculator@186c730
getHeight() Called
getWeight() Called
getBMI() Called
getWeight() Called
setWeight() Called
getHeight() Called
getWeight() Called
getBMI() Called
```

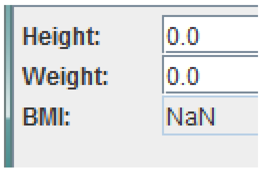
The general sequence of actions applies to any object following the Bean conventions.

NaN and Refreshing from the Program

To better understand the spreadsheet-like auto-update feature of ObjectEditor, consider the following code:

```
 public static void main (String[] args) {
   ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet();
   ObjectEditor.edit(bmiSpreadsheet);
   bmiSpreadsheet.setHeight(1.77);
   bmiSpreadsheet.setWeight(75);
 }
}
```

Figure ??? shows the ObjectEditor display after the setWeight() call.



Height: 0.0
Weight: 0.0
BMI: NaN

The height and weight are still the initial values, and the BMI is NaN – not a number – the result of dividing 0 by 0. It is as if ObjectEditor does not know that setHeight() and setWeight() were called after the edit() call!

ObjectEditor, indeed, does not know that these calls were made, as all we have told it is to display the object trough the edit() call. Our setters simply set the values of the corresponding instance variables, they do not communicate with ObjectEditor. ObjectEditor is a programmer-defined type and not part of the language – so it must be told when changes are made by code to the object it displays. The reason it was able to automatically update BMI in response to user edits to weight and height was that these changes were made through it – after every edit it took the sequence of actions mentioned above. When changes are not made through it, it must be told about them.

Later we will learn the ObjectEditor-independent techniques allowing an object to inform its observers about fine-grained changes it makes. For now, we will learn a simple ObjectEditor-specific call, refresh(), that simply tells it that something has changed in an object it is currently displaying. ObjectEditor responds to this call by redisplaying the whole object – that is, calling the getters of all properties of the object.

The following rewrite of the code above illustrates this call:

```
public static void main (String[] args) {
   ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet();
   OEFrame oeFrame = ObjectEditor.edit(bmiSpreadsheet);
   bmiSpreadsheet.setHeight(1.77);
   bmiSpreadsheet.setWeight(75);
   oeFrame.refresh();
  }
```

Unlike edit(), which is a call made on the class ObjectEditor itself, refresh() is called on an object whose interface is OEFrame. This object is returned by edit() and is an instance that represents the editor of the edited object. Thus, while edit() is an instance method, refresh() is an instance method. If a program does not make this call after an object sending, as an end-user, we can execute the Custom→refresh command, which makes the call on our behalf.

As we see here, the term ObjectEditor is being used for both the ObjectEditor class providing the edit ()method and also all the classes and interfaces such as OEFrame that are related to this call.

The refresh() method can be an important instrument for testing and demoing our objects. The goal of demoing or testing a program is to show the relationship between different input/output pair, that is, feed our code a series of input values and show the output for each input. In a Bean object, an input consists of a set of values for one or more of the independent properties, and the output consists of the associated value of the dependent computed values. ObjectEditor allows us to feed new values for the dependent properties without re-executing the code. To demo/test a Bean, we can write a main that executes the following algorithm:

1. Display the Bean using ObjectEditor.
2. Change a property to some value.
3. Select the property so that the person viewing the demo/text notices it.
4. Wait for the person to view the values of the dependent properties.
5. Go to 2.

This approach is illustrated in the code below:

```
 public static void main (String[] args) {
   ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet();
   bmiSpreadsheet.setHeight(1.77);
   bmiSpreadsheet.setWeight(75);
   OEFrame editor = ObjectEditor.edit(bmiSpreadsheet);
   ThreadSupport.sleep(5000);
   editor.select(bmiSpreadsheet, "Weight");
   bmiSpreadsheet.setWeight(70);
   editor.refresh();
   ThreadSupport.sleep(5000);
   editor.select(bmiSpreadsheet, "Height");
   bmiSpreadsheet.setHeight(0);
   editor.refresh();
   ThreadSupport.sleep(5000);
   editor.select(bmiSpreadsheet, "Weight");
   bmiSpreadsheet.setWeight(0);
   editor.refresh();  }
}
```

Here ThreadSupport.sleep() is a call (provided by ObjectEditor) to make the program "sleep" or do nothing for a certain number of milliseconds, which are specified as an argument to the call. The select() call highlights the property specified as its argument. The effect of this code, then, is to highlight some property so the user pays attention to it, change the property, and call refresh to display the results, multiple times, so that the user can see the display animating with different values of the properties, as shown in the following video: https://www.youtube.com/watch?v=dYfSuP3Io8I&feature=plcp . The user can then verify that each setter results in the expected value of BMI.

Silent Mistakes

We see above here that Bean conventions make code under code understandable to not only humans but also to ObjectEditor. ObjectEditor is not the only example of a tool interpreting these conventions. There are numerous other examples supporting a range of functionality including automatic testing, interactive user-interface composition, and Eclipse plugins. We will refer to a tool based on Beans or other programming conventions as convention-based tools.

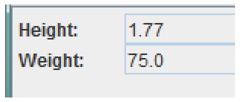
In such tools, conventions take the place of an API or style sheet in which a tool is explicitly told by the programmer how to find the names and types of the logical components of an object. All the programmers have to do is follow "intuitive" conventions, which arguably make sense also to human readers, rather than learn and use an API or some style sheet. The desired behavior is implicitly extracted from method headers.

This works well when no mistakes are made. When we make a mistake, in the case of an API or style sheet, the compiler, associated tools will hopefully give us enough information about the error through their behavior and messages. A convention-based tool will "fail" silently if it does not find the information we are trying to give it.

To illustrate, let us assume we change the class ABMISpreadsheet to the following:

```
public class ABMISpreadsheetNotFollowingBeanConventions  {
 double height = 1.77;
 double weight = 75;
 public double getWeight() {
  return weight;
 }
 public void set(double newWeight, double newHeight) {
  weight = newWeight;
  height = newHeight;
 }
 public double getHeight() {
  return height;
 }
 public void setHeight(int newHeight) {
  height = newHeight;
 }
 public double BMI() {
  return weight/(height*height);
 }
}
```

ObjectEditor creates the display shown in Figure ??? for this object.



Height: 1.77
Weight: 75.0

Neither height or weight is now editable, and perhaps worse, BMI is no longer displayed. Try to deduce the reason before reading about them below.

We made three mistakes in following the Bean conventions.

We named the function that returns the BMI as BMI() rather than getBMI(). As a result, ObjectEditor does not display the BMI property. Instead, it provides a menu item to invoke BMI().

We did not define a setWeight() method. We did define a set() method that allows both the weight and height to be changed, but is not a setter for either property. Thus, ObjectEditor considers the property readonly.

The reason why the Height property is readonly is perhaps the most subtle. Though setWeight() looks like a setter for Weight, its argument type, **int**, does not match the return type, **double**, of the getter(). The getter return type decides the type of the property.

As these mistakes are manifested in the display, they may be quickly discovered for those who understand Bean conventions well. In fact, for such programmers, ObjectEditor can serve as tool for checking if we have followed the intended conventions for humans and other tools.

Bean and Property Annotations

For those who cannot quite find the mistake in a header, ObjectEditor can serve as a trainer and a second pair of eyes. It can do so only if only if we become more explicit in what we are trying to do in our code. One mechanism for documenting our intention is to use comments, but these are neither parsable nor available at runtime to ObjectEditor. As we saw earlier, annotations do not have this problem. Hence, ObjectEditor supports several annotations to help us make our intention clear to both it and other humans.

Figure ?? shows the use of these annotations to express what we are doing with the class above:

```
import util.annotations.EditablePropertyNames;
import util.annotations.PropertyNames;
import util.annotations.StructurePattern;
import util.annotations.StructurePatternNames;
@StructurePattern(StructurePatternNames.BEAN_PATTERN)
@PropertyNames({ "Height", "Weight", "BMI"})
@EditablePropertyNames({"Height", "Weight"})
public class ABMISpreadsheetNotFollowingBeanConventions  {
      …

}
```

Different kinds of annotations are, in fact, programmer-defined class, and like other classes, must be imported if declared in another package. The three annotation classes we use here are defined by ObjectEditor, and thus must be imported from the associated packages. An instance of an annotation class is associated with a special "constructor" method whose arguments define its state. We can supply actual arguments to this method after the annotation name.

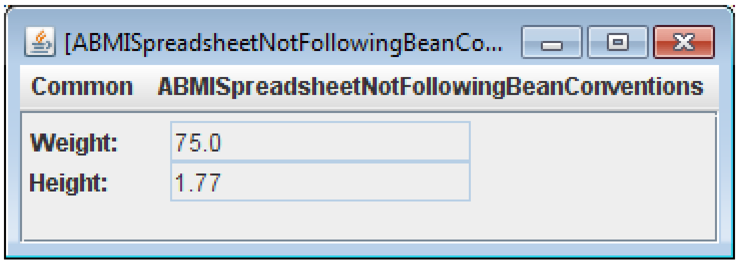
The StructurePattern annotations tells ObjectEditor which set of conventions are being followed in the annotated class, ABMISpreadsheetNotFollowingBeanConventions. Its argument `StructurePatternNames.BEAN_PATTERN ,` indicates that we are defining a Bean. The `PropertyNames` annotation takes an array argument whose elements indicate the names of the properties that the getter will define. Finally, the `EditablePropertyNames`, annotation gives the names of the properties that are editable, and thus, are expected to have setters.

Arguably, these annotations serve as documentation mechanism - by looking only at these annotations, the reader of the program gets an idea of the goal of the class. They are a higher-level mechanism than that provided by interfaces, as properties are synthesized from interfaces. Just as the compiler catches mistakes in not implementing an interface correctly, ObjectEditor catches mistakes in not implementing the declared properties correctly. Thus, in the example above, it will give the following error messages when asked to edit an instance of the class above:

```
E***For property: height in editable property names, please define a setter with the header:
        public void setHeight(double <parameter name>)
E***For property: weight in editable property names, please define a setter with the header:
        public void setWeight(double <parameter name>)
E***For property: BMI in  property names, please define a getter with the header:
        public <T> getBMI()
```

Display Order of Properties

As described above, the order of the properties does not matter in the two property annotations. There are times when want the properties to be displayed, not in the alphabetical order, but in an order decided by us. The PropertyName annotation does double duty as both an expression of the set of properties and the order in which properties are to be displayed. Thus, the display produced by adding it is the one shown in Figure ?? rather than the one we saw in Figure ???


[ABMISpreadsheetNotFollowingBeanCo...
Common
ABMISpreadsheetNotFollowingBeanConventions

Weight: 75.0
Height: 1.77

BMI is still not visible and the two properties are still uneditable. To fix these problems, we must fix the errors pointed out by ObjectEditor.

Making Properties Invisible

What if we define a getter for a property but do not declare it in either the PropertyNames or EditablePropertyNames list. For example, what if we rewrite the annotations code above as follows:

```
@StructurePattern(StructurePatternNames.BEAN_PATTERN)
@PropertyNames({ "Height", "BMI"})
@EditablePropertyNames({"Height"})
public class ABMISpreadsheetNotFollowingBeanConventions  {
      …

}
```

As before, the BMI property is not displayed because of the missing getter. Now the weight property is also not displayed because it appears in neither of the two lists. The resulting display is shown below.

The display of the weight property is not labeled as it is the only one displayed - there is no ambiguity as to what it displays.

There are times when we do want certain properties to be visible to other code but not to end-users. So hiding properties is not a bad idea. In case you made a mistake with the annotations, ObjectEditor shows the following warning:

What it is saying is that there is a more explicit way of hiding properties, the Visible annotation, which takes a Boolean argument, can be put before the getter of a property to hide it, as shown below:

```
@Visible(false)

public double getWeight() {

   ….

}
```

When we study composite objects, we will see that this annotation will be very useful – in particular for converting a “non tree” display structure to a “tree” structure.

Annotation vs. Efficiency

We see that the annotations above allow us to detect mistakes and influence the order of properties. They also help improve the efficiency of ObjectEditor. The StructurePattern annotation tells ObjectEditor to use Bean rather than other conventions to process method headers. The PropertyNames and EditablePropertyNames annotations tell s it which getters and setters to look. Thus it does not have to process all methods to extract property names. Given a set of property names, it simply needs to verify the associated methods exist.

Explanation Annotations → Tooltip Text

We saw above several benefits of replacing certain kinds of comments with annotations. Perhaps the most fundamental kind of comment is an explanation of some piece of code such as a class or method. The same or similar explanations are also sometimes provided as tooltips messages in the user-interface. It is possible to unite these two related concepts by replacing certain explanation comments with explanation annotations.

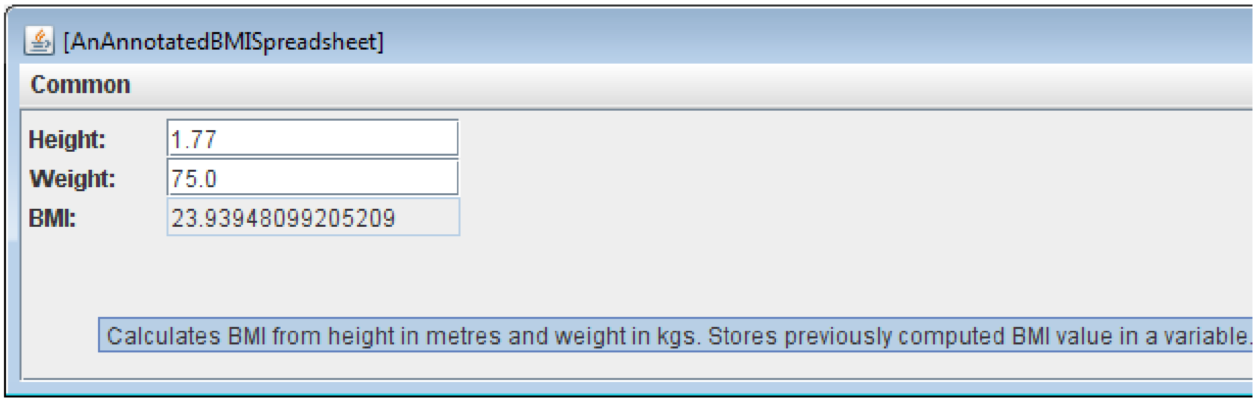
Consider the code fragment shown in Figure ???.

```
@Explanation("Calculates BMI from height in metres and weight in kgs.")
public interface AnnotatedBMISpreadsheet {
 public double getHeight();
 public void setHeight(double newVal);
 @Explanation("Weight is in Kilograms. ")
 public double getWeight();
 @Explanation("Changing weight automatically recalculates BMI.")
 public void setWeight(double newVal);
 public double getBMI();
}

@Explanation ("Stores previously computed BMI value in a variable.")
public class AnAnnotatedBMISpreadsheet implements
AnnotatedBMISpreadsheet
{…}
```

Here we have declared a BMI spreadsheet interface that has three Explanation annotations, which take String arguments. We have also declared a BMI spreadsheet class that implements this interface and provides an additional Explanation annotation.

Figure ??? and ??? shows how these annotation affect the user-interface of instances of the class above.


[AnAnnotatedBMISpreadsheet]

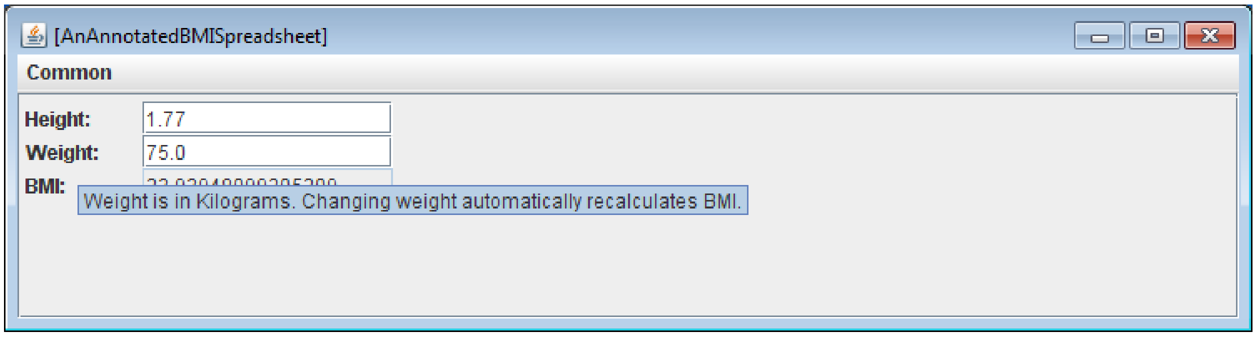
Common

Height: 1.77

Weight: 75.0

BMI: 23.93948099205209

Calculates BMI from height in metres and weight in kgs. Stores previously computed BMI value in a variable.


[AnAnnotatedBMISpreadsheet]

Common

Height: 1.77

Weight: 75.0

BMI:

Weight is in Kilograms. Changing weight automatically recalculates BMI.

When we hover over an area of the edit window of an object that is not a property display, the tooltip text shows a concatenation of the explanation annotation (argument) for the class and interface of the object. When we hover over a property display, the tooltip text shows a concatenation of the explanation annotations associated with its getters and setters (defined in the class and interface of the object).

What if both the interface and class define explanation annotations.

ObjectEditor can use the rules above to display arbitrary Beans, included nested Beans in which the properties of objects are themselves Beans. These rules, however, restrict the user-interface to nested forms of textual form items. How do we use ObjectEditor to create graphics?

ObjectEditor Graphics

The principle of asking ObjectEditor to edit objects still applies – however the displayed objects are now instances of special graphics types defined using graphics conventions understood by ObjectEditor. This means we must learn and use an additional set of conventions. Like Bean conventions, they are useful in their own right, as they make code easier to understand by readers of our programs. Unlike Bean conventions, however, these are not standard. Nonetheless, they should be intuitive as they are built on top of Bean conventions. In fact, we have already followed them in the definition of the geometric types we have discussed so far – point, line, rectangle, and oval. Let us first consider a point.

Point Graphics

Figure ??? shows how ObjectEditor displays an instance of ACartesianPoint. As we see in the figure, ObjectEditor understands that our object represents a geometric point, and uses the computer graphics coordinate system 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 ??? 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 ??? bottom-left). This command is also a toggle.

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

- If Its interface or class has the string “Point” in its name or it has a Point annotation discussed below.
- It has int properties, named X and Y, that represent its Cartesian coordinates.

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 coding classes and interfaces. We see here how these conventions are built on top of JavaBean conventions – they assume that the latter are followed in defining the X and Y properties.

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 ???.

The reason for using the name of the classes and interfaces of an object to decide if it a point is that there can be many classes with X and Y properties that are not points - in fact any geometric shape can be have such properties. It is possible for a class or interface of an object to not have the term Point in it and still not be a Point object, such as the interface below:

```
public interface CartesianPt {
        public int getX();
        public int getY();
}
```

If we wish ObjectEditor to treat this interface as a Point interface, we can preface its declaration with a Point annotation, which is a special case of the StructurePattern annotation we saw before that takes StructurePatternNames.*POINT_PATTERN* as an argument.

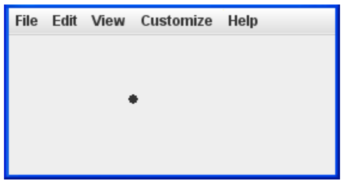
```
@StructurePattern(StructurePatternNames.BEAN_PATTERN)
public interface CartesianPt {
        public int getX();
        public int getY();
}
```

Even if our class or interface name follows the point annotation, it is a good idea to put this annotation for documentation and efficiency reasons.

It is possible for a class or interface to have the term Point in it and still not be a Point object, as illustrated by the following contrived example below:

```
public interface NotAPoint {
        public int getX();
        public int getY();
}
```

Based on the rules given so far, ObjectEditor will display an instance of this object as a point!

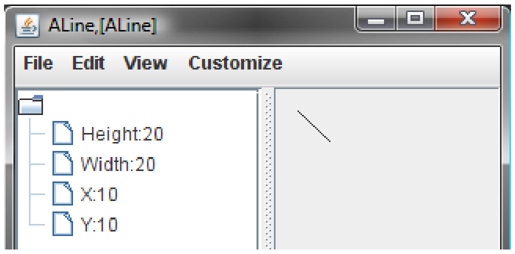


If this interface is intended to be some other kind of shape supported by ObjectEditor, then we can use the associated annotation. Otherwise we can use the annotation IsAtomicShape, giving it an argument of **false**.

```
@IsAtomicShape(false)
public interface NotAPoint {
        public int getX();
        public int getY();
}
```

Line, Oval, Rectangle

ObjectEditor defines a slightly more elaborate set of rules for recognizing lines, rectangles, and ovals. 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. For now, we will assume that the location is described using **int** properties, named X and Y, that give the Cartesian coordinates of the location. Later when we will see composite objects, we will extend these rules.



```
Height:20
Width:20
X:10
Y:10
```

There are many ways to distinguish between the three kinds of shapes described by these rules. The simplest approach supported by ObjectEditor assumes that the name of the class or interface of an object representing a line/rectangle/oval has the string Line/Rectangle/Oval in it. We were very careful in following these rules in ALine to ensure that instances of these classes are displayed as line shapes. Figure ??, showing both the tree and graphics view, illustrates that a Line instance is displayed as a line. In addition, ObjectEditor supports line, oval, and rectangle annotations, which are like the point and Bean annotation, but take different arguments:

```
@StructurePattern(StructurePatternNames.LINE_PATTERN)
@StructurePattern(StructurePatternNames.OVAL_PATTERN)
@StructurePattern(StructurePatternNames.RECTANGLE_PATTERN)
```

Of course, there are many useful logical representations of graphics objects that ObjectEditor currently does not support. For example, it does not support a representation in which a line is represented by its two end points.

Thus, ObjectEditor assumes an object represents a geometric rectangle/oval/line if:

- Its interface or class has the string “Rectangle”/”Oval”/”Line” in its name or it has a Rectangle/Oval/Line annotation discussed below.
- It has **int** X and Y properties, which are taken to be the coordinates of the upper left corner of the bounding box of the shape.
- It has **int** Width and Height properties, which are assumed to be the width and height of the bounding box.

Strings and Images

Most useful graphics displays also display text and images – often to label other shapes. Like other bounded shapes we have seen so far, they can be characterized by a bounding box. The location of this box must of course be specified by the programmer. The size of the box, on the other hand, can be computed automatically computed from the string or image. Sometimes, the programmer may also wish to specify the size, so that the string or image is clipped to a maximum size.

ObjectEditor assumes an object represents a geometric String shape if:

- Its interface or class has the string "String" in its name or it has a String annotation discussed below.
- It has a String Text property, which is taken to hold the string to be displayed.
- It has **int** X and Y properties, which, as in the case of the other bounded shapes, are expected to be the coordinates of the upper left corner of the bounding box of the shape.

Similarly, an object represents a geometric an image shape if:

- Its interface or class has the string "Image" in its name or it has an image annotation discussed below.
- It has **int** X and Y properties, which, as above, are assumed to be the coordinates of the upper left corner of the bounding box of the shape.
- It has an ImageFile property, which is taken to be the (JPG, PNG, …) file to be displayed in the bounding box.

Both an image and string shape can have additional optional Height and Weight properties. If present, they decide the size of the bounding box of the shape. Parts of the shape that do not fit in the bounding box are clipped, that is, not displayed. If these properties are not present, then the size is made the same as the size of the string/image.

As with other shapes, the string and shape pattern are defined using associated constants:

```
@StructurePattern(StructurePatternNames.STRING_PATTERN)
@StructurePattern(StructurePatternNames.IMAGE_PATTERN)
@StructurePattern(StructurePatternNames.STRING_PATTERN)
```

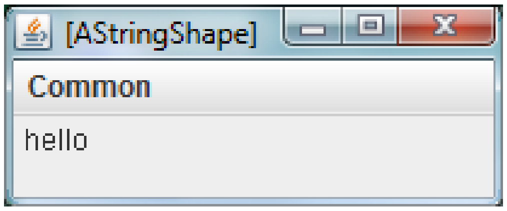
The following is an example of a class defining a string shape.

```
@StructurePattern(StructurePatternNames.STRING_PATTERN)
public class AStringShape implements StringShape {
  String text;
  int x, y;
  public AStringShape(String initText, int initX, int initY) {
    text = initText;
    x = initX;
    y = initY;
  }
  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 getText() {return text;}
  public void setText(String newVal) {text = newVal;}
}
```

The following main method displays an instance of this class.

```
public static void main(String args[]) {
 StringShape hello= new AStringShape("hello", 0, 0);
 ObjectEditor.edit(hello);
}
```

Figure ??? shows the display created by the main method:



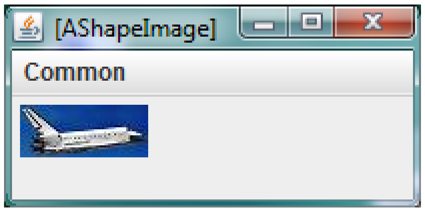
An example of a class describing an image is given below:

```
@StructurePattern(StructurePatternNames.IMAGE_PATTERN)
public class AShapeImage implements ImageShape {
  String imageFileName;
  int x, y;
  public AShapeImage (String initImageFileName, int initX, int initY) {
    imageFileName = initImageFileName;
    x = initX;
    y = initY;
  }
  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 void setImageFileName(String newVal) {imageFileName = newVal ;}
}
```

Example code instantiating the class:

```
public static void main (String args[]) {
    ImageShape shuttle = new AShapeImage("shuttle2.jpg", 0, 0);
    ObjectEditor.edit(shuttle);
}
```

The display created by the code:



An example of a class describing an image is given below:

Mutable Shapes

Unlike a point, the other shapes can have editable properties, as the code examples above show. If we change these properties and inform ObjectEditor about the changes, by for instance, calling refresh, then ObjectEditor will change the display accordingly. For example, if we change the ImageFileName property of an image shape, and call refresh(), ObjectEditor will display the new image at the original location. We can of course change the location and size of each of these shapes to move or resize them.

Object-based Graphics vs. API-based Graphics

Some of you may have used predefined features in Java to display shapes. We will study these later in much more depth. For now, here is a quick superficial discussion of the differences to help you better understand the approach used here.

Suppose we wish to draw a line using ObjectEditor. If we have not done so already, we create a class such as ALine that follows the line shape conventions. Next, we call the ObjectEditor edit method to display the line, passing in values of the X, Y, Width and Height properties.

```
ObjectEditor.edit(new ALine(x, y, w, h));
```

To draw a line using Java's native features seems simpler - we make a call of the form:

```
graphics.drawLine(x1, y1, x2, y2)
```

Here, the graphics variable holds an object defined by Java, and arguments to the drawLine() method specify the two end points of the line. A minor difference, of course, is that we represent the line differently in the two systems, further showing the possible of alternative solutions to a problem. The more interesting difference is that under the Java approach, each time we wish to redraw a shape, we must make a call to drawLline() and other API drawing calls. In contrast, ObjectEditor, requires the use of an editable shape object. Each time we wish to redraw the shape, we change the properties of the object. Associating an object with a shape (displayed through ObjectEditor or other code) has several advantages. Code that needs to manipulate the shape can simply change the object - there is no need to write or communicate with user-interface code that displays the actual shape. Moreover, it is possible to display the object using in ways by, for instance, creating a tree view for it, in addition to, or in place of the graphics view. The MVC discussion will make this much clearer. For now, the main difference is that, if such a class does not already exist, we create a class for each kind of shape we display, which is not a bad idea if we are trying to learn how to create classes. For those who gain nothing from this exercise, ObjectEditor does provide predefined classes for all of the shapes Java can draw directly.

One apparent problem with the ObjectEditor approach is that we cannot draw multiple shapes at the same time. What if we wanted an image labeled by a String? Java will allow us to make multiple draw calls from our drawing code. As we will see next, when we study composite objects, ObjectEditor will require us to define a “tree” object structure whose “leaves” are the “atomic” shapes it (and Java) support.

Customizing ObjectEditor Window

ObjectEditor provides an API to determine the size of the top-level window and the panels in this window. The ObjectEditor.edit() method is a function that returns an instance of type OEFrame. This type defines methods to resize the method, and hide and show the drawing, tree, and main panels, as shown below:

```
OEFrame frame = ObjectEditor.edit(new ALine (0, 0);

frame.hideDrawPanel();

frame.showTreePanel();

frame.setSize(500, 400); // sets the width and height of window
```

Summary

- Java objects, methods, and classes correspond to real-world objects, the operations that can be performed on them, and the factories that manufacture the objects, respectively.
- A class definition consists of a class header identifying its name and accessibility; and a class body declaring a list of methods.
- A method definition consists of a method header identifying its name, accessibility, formal parameters, and return type; and a method body specifying the statements to be executed when the method is called.
- Formal parameters of a method are variables to which actual parameters are assigned when the method is called.
- The set of values that can be assigned to a formal parameter/returned by a method is determined by the type of the parameter/method.
- The process of writing and executing code we have seen so far consists of declaring one or more methods in a class, compiling the class, asking the interpreter to start ObjectEditor, asking ObjectEditor to instantiate the class, and finally, invoking methods on the new instance.
- By following style principles in writing code, we make the code more understandable to us, others, and tools such as ObjectEditor.

Other Resources

Code package: lectures.objects

Slides: PowerPoint PDF

Exercises

1. Define class, method, object, variable, formal parameter, actual parameter, type, statement, return statement, and expression.
2. What are the case conventions for naming classes, methods, and variables?
3. Define syntax error, semantics error, logic error, accessibility error, and user error.
4. Why is it useful to follow style principles?
5. Suppose we need a function that converts an amount of money in dollars to the equivalent amount in cents and can be invoked by ObjectEditor. The following class attempts to provide this function Identify all errors and violation of case conventions in the class.

```
class aMoneyconverter {

 int Cents (dollars) {

return dollars*100;

}

}
```

6. Write and test a function, `fahrenheit()`, that takes a parameter an integer centigrade temperature and converts it into the equivalent integer Fahrenheit temperature, as shown in the figures below. Assuming F and C are equivalent Fahrenheit and centigrade temperatures, respectively, the conversion formula is:

F = C * 9/5 + 32

Implement the function in the class `ATemperatureConverter.` Use a bare-bones programming environment to develop and execute the class.