Chapter 16

Using Inheritance and Polymorphism

In this chapter we make use of inheritance and polymorphism to build a useful data structure.

16.1 Abstract Classes

Circle1a (16.1) is a variation of Circle1 (7.1) with the circumference() method renamed perimeter().

```java
public class Circle1a {
    private final double PI = 3.14159;
    private double radius;
    public Circle1a(double r) {
        radius = r;
    }
    public double perimeter() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 16.1: Circle1a—a slightly modified Circle1

Consider a new class—Rectangle1a (16.2). (The “1a” in its name simply implies that it is somehow associated with the Circle1a class.)

```java
public class Rectangle1a {
    private double width;
    private double height;
    public Rectangle1a(double w, double h) {
```

13 March 2008 Draft © 2008 Richard L. Halterman
width = \( w \);
height = \( h \);
}\n
public double area() {
    return width * height;
}\n
public double perimeter() {
    return 2*width + 2*height;
}\n
Listing 16.2: \texttt{Rectangle1a}—a stateful rectangle

Objects of these classes represent circles and rectangles—simple two-dimensional geometric shapes. We say they are \textit{kinds of} shapes; that is, a circle is a \textit{kind of} shape, and a rectangle is another \textit{kind of} shape. As such, these specific kinds of shapes exhibit some common characteristics; for example, they both have areas and perimeters. We can \textit{classify} these specific circle and rectangle shapes under the general category of \textit{shapes}.

Suppose we want to write a method that would accept any kind of shape object and use the methods common to all shape objects to compute some useful result; for example, the perimeter-to-area ratio of an object, as in \texttt{PerimeterToAreaRatio} (\textit{16.3}).

public class PerimeterToAreaRatio {  
    public static double perimeterToAreaRatio(Shape s) {
        return s.perimeter()/s.area();  
    }  
}  

Listing 16.3: \texttt{PerimeterToAreaRatio}—computes the perimeter-to-area ratio of a geometric shape

Observe that if we replace \texttt{Shape} in the parameter list of \texttt{perimeterToAreaRatio()} with \texttt{Circle1a} objects, it will work for \texttt{Circle1a} objects. Similarly, if we replace \texttt{Shape} with \texttt{Rectangle1a}, it will work for \texttt{Rectangle1a} objects. We do not want to write two different \texttt{perimeterToAreaRatio()} methods, one for each shape type. Instead, we really want only one method that works on both shape types, as shown here in the \texttt{PerimeterToAreaRatio} class. Inheritance provides the way the make this happen.

In OOP, we use \textit{class hierarchies} formed by inheritance relationships to classify kinds of objects. A Java class represents a kind of an object. When two classes share properties and are related to each other conceptually, we often can create a a new class exhibiting those common properties. We then modify the two original classes to make them subclasses of the new class. In our example, circles have areas and perimeters, and rectangles have areas and perimeters. Both are kinds of shapes, so we can make a new class named \texttt{Shape} and make \texttt{Circle1a} and \texttt{Rectangle1a} subclasses of \texttt{Shape}.

How do we capture the notion of a general, nonspecific shape in a Java class? Here are our requirements:

- We would name the class \texttt{Shape}.  

13 March 2008 Draft © 2008 Richard L. Halterman
• Any object that is a kind of Shape should be able to compute its area and perimeter through methods named area() and perimeter(). (Circle class and Rectangle class object both qualify.)

• We cannot write actual code for the area() and perimeter() methods in our generic Shape class because we do not have a specific shape. No universal formula exists to compute the area of any shape because shapes are so different.

• Clients should be unable to create instances of our Shape class. We can draw a circle with a particular radius or a rectangle with a given width and height, but how do we draw a generic shape? The notion of a pure shape is too abstract to draw. Likewise, our Shape class captures an abstract concept that cannot be materialized into a real object.

Java provides a way to define such an abstract class, as shown in Shape (16.4).

```java
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();
}
```

Listing 16.4: Shape—a generic shape

In Shape (16.4):

• The class is declared abstract. Such a class is called an abstract class. We are unable to create instances of an abstract class, just as we really cannot have a plain shape object that is not some kind of specific shape.

• An abstract class may have zero or more methods that are declared abstract. Abstract methods have no bodies; a semicolon appears where the body of a non-abstract method would normally go. We are declaring the method but not defining it. Since we have no concrete information about a generic shape we cannot actually compute area and perimeter, so both of these methods are declared abstract.

• While not shown here, an abstract class may contain concrete (that is, non-abstract) methods as well. An abstract class can have non-abstract methods, but a non-abstract class may not contain any abstract methods.

A non-abstract class is called a concrete class. All classes are either abstract or concrete. All the classes we considered before this chapter have been concrete classes. A non-abstract method is called a concrete method. A concrete class may not contain abstract methods, but an abstract class may contain both abstract methods and concrete methods.

Given our Shape class we can now provide the concrete subclasses for the specific shapes in Circle (16.5) and Rectangle (16.6).

```java
public class Circle extends Shape {
    private final double PI = 3.14159;
    private double radius;
    public Circle(double r) {
        radius = r;
    }
}
```

13 March 2008 Draft © 2008 Richard L. Halterman
16.1. ABSTRACT CLASSES

```java
public double perimeter() {
    return 2 * PI * radius;
}
public double area() {
    return PI * radius * radius;
}
```

Listing 16.5: Circle—a circle subclass

```java
public class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double w, double h) {
        width = w;
        height = h;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * width + 2 * height;
    }
}
```

Listing 16.6: Rectangle—a rectangle subclass

Now PerimeterToAreaRatio (16.3) works as is on any Shape subclass. Polymorphism enables the perimeterToAreaRatio() to execute the proper area() and perimeter() code for the object at hand. Figure 16.1 shows the UML diagram for our shapes class hierarchy. Abstract class names are shown in italics in the UML class diagrams.

```
Shape
  ↓
Circle
  ↓
Rectangle
```

Figure 16.1: Class hierarchy of shape classes. It is common practice to omit the superclass of Shape; that is, the java.lang.Object class is not shown here.

Suppose we must develop an inventory program for a produce market. One of the market’s specialties is fresh fruit, including bananas and several varieties of apples and grapes. For example, we might have a class for golden

13 March 2008 Draft © 2008 Richard L. Halterman
delicious apples and another for granny smith apples. We note that both varieties have some common properties (after all, they are both kinds of apples). It makes sense to create a common superclass, Apple, and derive the GoldenDelicious and GrannySmith classes from Apple, specializing the subclasses as required. Classes for red and white grapes could have Grape as a common superclass. Further, all fruit have some common properties, like color (red, yellow, etc.), taste (sweet, tangy, etc.), and size (volume or mass). Given this scenario we can create a hierarchy of classes as illustrated by the UML diagram in Figure 16.2. Ordinarily, we use classes as a blueprint for creating objects. We note that while golden delicious apples and granny smith apples would have a place in the inventory database, generic “apples” would not.

To recap, when two or more classes have common characteristics, and the classes conceptually represent specific examples of some more general type, we can use inheritance to factor out their common properties. We do so as follows:

- Make a new superclass that contains the common characteristics.
- Declare the class abstract if it represents an abstract concept and actual objects of this class do not make sense because there is not enough specific information to implement all of its required functionality.
- Modify the existing specific classes to be subclasses of the new superclass.

The new superclass is meant to distill the common characteristics of its subclasses. It represents the abstract notion of what it means to be instances of any of its subclasses.

### 16.2 A List Data Structure

Armed with our knowledge of inheritance, polymorphism, Java’s wrapper classes (§ 13.4), and abstract classes, we can build a versatile list data structure. First, we need to think about the nature of lists:

- A list holds a collection of items in a linear sequence. Every nonempty linear sequence has the following properties:
  - there exists a unique first item in the list
  - there exists a unique last item in the list

![Figure 16.2: Class hierarchy of fruit classes.](image-url)
– every item except the first item has a unique predecessor
– every item except the last item has a unique successor

• An empty list contains nothing. One empty list is indistinguishable from any other empty list. It follows that there should be exactly one empty list object, because making more than one does not provide any advantages.

• We can view any nonempty list as the first item followed by a list (the rest of the list). A nonempty list has at least one item. In a list containing only one item, the rest of the list refers to the empty list.

• It is easy to determine the number of items in a list:
  – If the list is empty, the number of items it contains is zero.
  – If the list is nonempty, the number of items it contains is one (for its first item) plus the number of items in the rest of the list.

As we shall see, this recursive description is readily translated into a recursive method.

• It is easy to add an item to the end of a list:
  – If the list is empty, we just make a new nonempty list whose first item is the item we wish to add. The rest of this new list is the empty list.
  – If the list is nonempty, we simply add the new item to the rest of this list.

This recursive algorithm also is easily translated into Java code.

As you can see, we have two distinct kinds of lists: empty lists and nonempty lists. We have to be able to treat both empty and nonempty lists as just lists even though empty lists and nonempty lists are functionally different kinds of objects. We need to distill the properties common to all kinds of lists into an abstract list class. From this abstract list class we will derive the concrete subclasses for empty and nonempty list objects. Figure 16.3 shows the resulting class hierarchy.

```
List (16.7) provides good starting point:

public abstract class List {
    public abstract int length();
    public abstract List append(Object newElement);
    public String toString() {
```

Figure 16.3: Class hierarchy of list classes.
As we can see in `List` (16.7):

- Any list object can determine its length; that is, the number of elements it holds.
  
  ```java
  public abstract int length();
  ```

  We cannot specify how this method should work since empty and nonempty lists will do this differently; therefore, this method is abstract.

- New items can be appended to any list object.
  
  ```java
  public abstract List append(Object newElement);
  ```

  Again, we do not have enough detail in a generic list to indicate exactly how this is to be done, so `append()` is declared abstract. The parameter type of `append()` is `Object`. Since all classes are subclasses of `Object` (§ 14.1), either directly or indirectly, this means any reference type can be added to our list. Since all the primitive types have wrapper classes (§ 13.4), any primitive type as well can be added to our list. This means anything can be added to a list, even another list!

  The return type is `List`, but when a subclass overrides the `append()` method it will actually return a reference to an instance of a subclass of `List`. Because of the `is a` relationship, this is legal. For all we know now, the `append()` method returns a reference to some kind of list object.

- Every list object can be rendered as a string in the same way: The first character is `[` and the last character is `]`. The characters in between are determined by another method, `toStringHelper()`. Since actual code can be written for the `toString()` method, this method is concrete, not abstract.

  ```java
  return "[" + toStringHelper() + "]";
  }
  protected abstract String toStringHelper();
  ```

  The `toStringHelper()` method is abstract, since for a generic list object we cannot know how to access its contents, if indeed it has any contents. It uses a new access privilege—`protected`. A `protected` member is accessible to code in subclasses and classes within the same package, but inaccessible to all other classes. This is exactly the level of protection we need here. `toStringHelper()` is meant to be an internal helper method; it is not meant to be used by clients. Making it `private`, however, would make it inaccessible to subclasses, meaning that subclasses could not override it. Since it is declared abstract, concrete subclasses must override it. The `protected` specifier essentially means `public` to subclasses and classes within the same package and `private` to all other classes.

Since `List` is abstract, we cannot create actual `List` objects. `List` serves as the superclass for two concrete subclasses that we will use to build real lists. The first is `EmptyList` (16.8):
16.2. A LIST DATA STRUCTURE

Despite its small size, EmptyList (16.8) is full of interesting features:

- EmptyList is a subclass of List,

```java
public class EmptyList extends List {
    // This constant is used when an empty list is needed
    public static final List EMPTY = new EmptyList();

    private EmptyList() {} // Cannot create an instance of EmptyList

    public int length() {
        return 0;
    }
    public List append(Object newElement) {
        return new NonemptyList(newElement, EMPTY);
    }
    protected String toStringHelper() {
        return " ";
    }
}
```

Listing 16.8: EmptyList—Class for empty list objects

- A class constant consisting of an EmptyList instance

```java
public static final List EMPTY = new EmptyList();
```

is made available to clients for general use. As indicated in the next item, any reference to an empty list must use this constant.

- The only constructor is private.

```java
private EmptyList() {} // Cannot create an instance of EmptyList
```

As the comment says, this means clients are not allowed to create EmptyList objects directly using the new operator. For example, the statement

```java
EmptyList e = new EmptyList(); // Compiler error!
```

appearing anywhere outside of this class will result in a compiler error. This fact coupled with the EMPTY EmptyList constant means that exactly one EmptyList object will ever exist when this class is used. We say that EMPTY is a singleton empty list object.
• Empty lists trivially have no elements:

```java
public int length() {
    return 0;
}
```
so the length of an empty list is always zero.

• Appending a new item to an empty list results in a new nonempty list:

```java
public List append(Object newElement) {
    return new NonemptyList(newElement, EMPTY);
}
```

We will see the `NonemptyList` class next. Its constructor takes two arguments:

– The first argument is the first item in the list.
– The second argument is the rest of this new nonempty list.

Observe that the net effect is to make a list containing only one element—exactly what we need when appending to an empty list.

• Converting the contents of the empty list to a string is also trivial.

```java
protected String toStringHelper() {
    return " ";
}
```
The `toStringHelper()` method simply returns a space—there are no elements to show.

• The `toString()` method is inherited from `List`. It ensures the contents of the resulting string are bracketed within `[]`. This superclass `toString()` method calls `toStringHelper()` polymorphically, so the correct `toStringHelper()` is called for the exact type of the list object.

Even though `EmptyList` is a concrete class, we cannot use it until we implement `NonemptyList (16.9):

```java
public class NonemptyList extends List {
    private Object first;
    private List rest;

    public NonemptyList(Object f, List r) {
        first = f;
        rest = r;
    }

    public int length() {
        return 1 + rest.length();
    }
}
```
public List append(Object newElement) {
    return new NonemptyList(first, rest.append(newElement));
}

protected String toStringHelper() {
    return " * + first + rest.toStringHelper();
}

Listing 16.9: NonemptyList—Class for nonempty list objects

In NonemptyList (16.9):

• Two private fields:

    private Object first;
    private List rest;

capture our original definition of what constitutes a nonempty list:

– Every nonempty list has a first item (first), and
– Every nonempty list has the rest of the list (rest) that follows the first item.

• The type of first is Object. The practical result is any type of data can be assigned to the first instance variable.

• The type of rest is List. This means references to both EmptyList objects and NonemptyList objects can be assigned to the rest instance variable. A list that contains only one item would assign the item to first and then assign rest to the empty list.

• The constructor initializes the instance variables:

    public NonemptyList(Object f, List r) {
        ...
    }

• The length() method

    public int length() {
        return 1 + rest.length();
    }

works exactly as we originally specified. The length of a nonempty list is one (for its first item) plus the length of the rest of the list. Note the recursive nature of this method. It is up to the remainder of the list (referenced by rest) to compute its own length. Empty lists will report zero, and nonempty lists call this same method polymorphically but with a different object.

• The append() method
16.2. A LIST DATA STRUCTURE

```java
public List append(Object newElement) {
    return new NonemptyList(first, rest.append(newElement));
}
```

simply creates a new list object. We know for sure the type of list to create is a NonemptyList since we are appending an item onto the current (nonempty) list. The first item in the new list will be the same as the first item in this list. The rest of the new list is the result of appending the new item to the end of the current rest of the list. The recursive nature of the method call leads eventually to an attempt to append the new item to the empty list. The recursion stops there (the append() method in EmptyList is definitely nonrecursive), and a new nonempty list containing the new item is returned.

- The toStringHelper() method

  ```java
  protected String toStringHelper() {
    return "" + first + rest.toStringHelper();
  }
  ```

  is another recursive procedure. We append the first item and then defer the rest of the task to the rest object.

- As with EmptyList, the toString() method is inherited from List. When this superclass toString() method is called on behalf of a NonemptyList object, toString() calls NonemptyList's toString-Helper() polymorphically.

We now have everything in place to experiment with our new list data structure:

```text
Welcome to DrJava. Working directory is /Users/rick/java
> e = new EmptyList(); // Attempt to create an empty list
IllegalAccessException: Class koala.dynamicjava.interpreter.context.GlobalContext can not access a member of class EmptyList with modifiers "private"
    at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:486)
> e = EmptyList.EMPTY; // Do it the intended way
> e
[ ]
> list1 = e.append("Fred");
> list1
[ Fred ]
> list2 = list1.append("Wilma");
> list2
[ Fred Wilma ]
> list3 = list2.append(19.37);
> list3
[ Fred Wilma 19.37 ]
> list4 = list3.append(15);
> list4
[ Fred Wilma 19.37 15 ]
> list5 = list4.append("Barney");
> list5
[ Fred Wilma 19.37 15 Barney ]
```

13 March 2008 Draft © 2008 Richard L. Halterman
Figure 16.4 illustrates the data structure referenced by list5 as created by this sequence of interactions. If the

structure of list list5 is represented by the following:

```
list5 = "Fred" → "Wilma" → 19.37 → 15 → "Barney" → e
```

then the activity performed by the call list5.length() is
16.2. A LIST DATA STRUCTURE

The compiler automatically creates the appropriate wrapper objects for the primitive types 19.37 and 15.

Like Java’s String objects, our list objects are immutable. Clients cannot modify the contents of a list. Our append() method does not modify an existing list; it creates a new list with the new item added on the end. The original list is unaffected by the action. What if we wish to modify a list? We can simply reassign the list reference, as is:

```
list5 = list5.append(2);
```

In this case the original list of list5 is replaced by the new list created by the append() method call.

As an interesting side note, since our lists can hold any kinds of objects, we can easily have lists of lists:

```
Welcome to DrJava. Working directory is /Users/rick/java
> list = EmptyList.EMPTY;
> list
[ ]
> list = list.append(19.37);
> list
[ 19.37 ]
> list = list.append("Fred");
> list = list.append("Wilma");
> list
[ 19.37 Fred Wilma ]
> list.length()
3
> list = list.append(list);
> list
[ 19.37 Fred Wilma [ 19.37 Fred Wilma ] ]
> list.length()
4
```
The length of the whole list at the end is four since the last element (itself a list) is a single object (a `NonemptyList` object).

ListTest (16.10) provides some lightweight testing for our list data structure.

```java
import junit.framework.TestCase;

generic class ListTest extends TestCase {

    public void testAppendMethod() {
        List list = EmptyList.EMPTY;
        assertEquals("[ ]", list.toString());
        assertEquals(0, list.length());
        list = list.append(22);
        //System.out.println(list);
        assertEquals("[ 22 ]", list.toString());
        assertEquals(1, list.length());
        list = list.append("Fred");
        assertEquals("[ 22 Fred ]", list.toString());
        assertEquals(2, list.length());
        list = list.append(2.2);
        assertEquals("[ 22 Fred 2.2 ]", list.toString());
        assertEquals(3, list.length());
        list = list.append("Wilma");
        assertEquals("[ 22 Fred 2.2 Wilma ]", list.toString());
        assertEquals(4, list.length());
    }
}
```

Listing 16.10: ListTest—JUnit test file

To conclude, it is important to note that the immutable list implementation presented here is simple and elegant but not very efficient. A nonrecursive, mutable version presented in § 18.4 is more efficient. We will consider this more efficient implementation after we cover iteration in Chapter 17.

### 16.3 Interfaces

Java provides a construct that is similar to an abstract class with methods that are exclusively `public` and `abstract`. An interface specifies a collection of methods that any class that is to comply with that interface must implement. Since `Shape` (16.4) is an abstract class containing no instances variables and no concrete methods, it can be expressed as an interface, as shown in `IShape` (16.11).

```java
public interface IShape {
    double area();
    double perimeter();
}
```
Listing 16.11: IShape—an interface for shape objects

From the IShape interface we can now specify that particular classes comply with the interface using the implements keyword, as shown in Ring (Listing 16.12) and Quadrilateral (Listing 16.13).

```java
public class Ring implements IShape {
    private final double PI = 3.14159;
    private double radius;
    public Ring(double r) {
        radius = r;
    }
    public double perimeter() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 16.12: Ring—a ring (circle) class implementing the shape interface

```java
public class Quadrilateral implements IShape {
    private double width;
    private double height;
    public Quadrilateral(double w, double h) {
        width = w;
        height = h;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * width + 2 * height;
    }
}
```

Listing 16.13: Quadrilateral—a quadrilateral (rectangle) class implementing the shape interface

The Ring and Quadrilateral classes both implement the IShape interface. They must contain at least the methods defined in IShape, but implementing classes can add additional methods and add fields.

Methods defined in interfaces are implicitly abstract and public; thus, area() and perimeter() within IShape are public abstract methods. An interface is like an abstract class in that it is not possible to create instances of interfaces. Such a statement is illegal:

13 March 2008 Draft © 2008 Richard L. Halterman
an interface cannot contain instance variables or general class variables but can define constants (final static fields).

An interface serves as a contract that classes must fulfill that claim to comply with that interface. A class declares that it implements an interface with the implements keyword followed by a list of interfaces with which it complies. Unlike subclassing, where a single class cannot have more than one superclass, a class may implement multiple interfaces. In fact, any programmer-defined class will definitely extend a class (Object, if nothing else) and also may implement one or more interfaces.

Like a class, an interface defines a type; an implementing class is a subtype of the interface, just like a subclass is a subtype of its superclass. A special relationship exists between an interface and a class that implements that interface. This relationship is a can do relationship because an interface expresses what can be done in terms of method signatures. The interface specifies what can be done but cannot express how it is to be done. The how can only be specified in a method body, but in an interface all methods must be abstract. In practice, however, the can do relationship behaves like the is a relationship; that is, an object that is a subtype of an interface can be used in any context that expects the interface type. ShapeReport (Listing 16.14) shows how this subtyping relationship works.

```java
public class ShapeReport {
    public static void report(IShape shape) {
        System.out.println("Area = " + shape.area()
            + ", perimeter = " + shape.perimeter());
    }
    public static void main(String[] args) {
        IShape rect = new Quadrilateral(10, 2.5),
            circ = new Ring(5.0);
        report(rect);
        report(circ);
    }
}
```

Listing 16.14: ShapeReport—Illustrates the is a relationship for interfaces

Notice two things in ShapeReport:

- The report() method in ShapeReport expects an IShape type and works equally well with Quadrilateral and Ring objects. This is because of the can do relationship between an interface and its implementing class.

- In main(), the declared type of both rect and circ is IShape. While a literal IShape object cannot be created, a Quadrilateral object can be assigned to an IShape reference. Likewise, a Ring object can be assigned to an IShape reference. Again, this is because of the can do relationship between the implementing classes and IShape.

The reason that the parameter passing and assignment works is that, just as a subclass reference can be assigned to a superclass reference, a reference to a subtype can be assigned to a supertype reference. In fact the subclass-superclass relationship is a special case of the subtype-supertype relationship: the superclass of a class is its supertype, and a subclass of a class is a subtype of that class.
16.4 Summary

- A Java class represents a kind of an object, and a common superclass is used to organize different kinds of objects with common characteristics into the same category.
- An abstract method has no body.
- Concrete subclasses must override inherited abstract methods.
- A concrete class may not contain any abstract methods.
- An abstract class contains zero or more abstract methods.
- An abstract class may contain concrete methods.
- In a UML diagram, the name of an abstract class is italicized, and the name of a concrete class is not italicized.

16.5 Exercises

1. What reserved word specifies a class to be abstract?
2. Can instances be made of an abstract class?
3. Can all the methods in an abstract class be concrete?
4. Can any methods in a concrete class be abstract?
5. In Figure 16.2, which classes are abstract and which are concrete?
6. Explain how the append() method in the NonemptyList class correctly adds new elements to the end of the list.
7. Add the following methods to the List class and each of its subclasses as necessary:
   
   (a) public List prepend(Object newElement) — inserts a new element onto the front of the list
   (b) public List concat(List other) — splits two lists together in a manner analogous to string concatenation.
   (c) public boolean contains(Object seek) — returns true if seek is found in the list; otherwise, returns false. The equals() should be used to check for equality.
   (d) public boolean equals(List other) — determines if two lists contain the same elements (using their equals() methods) in the same order.
   (e) public Object get(int index) — returns the item in position index. The first element in a nonempty list is at index zero. The method should return null if an invalid index is given:
      • Any index is invalid for an empty list.
      • In a nonempty list with n elements, a valid index is in the range 0 ≤ index < n.
8. Use list object to record the states that a traffic light assumes. This will keep a log of all the states that a traffic light enters. Make a new traffic light type named LoggedTrafficLight, a subclass of TextTrafficLight (10.2). Every call to its setState() method should append the requested state to the end of the list. Add a method named reviewLog() that simply prints out, in order, the states assumed by the traffic light since its creation. You should use but not modify the code of TrafficLightModel (10.1) and TextTrafficLight (10.2).