M301: Software Systems & their Development

Block 1: Introduction to Java

Unit 4: Inheritance, Composition and Polymorphism

Aims of the unit:

- Study and use the Java mechanisms that support reuse, in particular, inheritance and composition;
- Analyze some of the implications of polymorphism for an object-oriented language, such as Java;
- Explore various forms of polymorphism.

Block 1: Introduction to Java

Unit 4: Inheritance, Composition and Polymorphism

Section 1: Java Mechanisms for Software Reuse

This section introduces you to the two main mechanisms for building new classes from existing classes: inheritance and composition.

Objectives:

On completing this section you should be able to:

- Compare and contrast the inheritance and composition mechanisms, and be able to analyze the main advantages and disadvantages of each in a particular situation;
- Use the knowledge gained by completing objective 1 as a guide to building classes using either inheritance or composition, or both;
- Describe how the *is-a* and *has-a* relationships relate to inheritance, subtyping and composition.

Summary of Chapter 10

Object-oriented programming makes possible a level of *software reuse* that is orders of magnitude more powerful than that permitted by previous software construction techniques.

Inheritance in Java is made more flexible by the presence of two different mechanism, interfaces and subclassification.

Substitutability

The concept of substitutability is fundamental to many of the most powerful software development techniques in OO programming.

Recall that substitutability referred to the situation that occurs when a *variable* declared as one type is used to hold a *value* derived from another type.

In Java, substitutability can occur either through the use of classes and inheritance, or through the use of interfaces (as we have already seen in the previous examples).

The Is-a *Rule and the* Has-a *Rule*

The *is-a* relationship holds between two concepts when the first is a specialized instance of the second.

That is, for all practical purposes the behavior and data associated with the more specific idea form a subset of the behavior and data associated with the more abstract idea.

For example, a Dog *is-a* Mammal, a PinBall *is-a* Ball, and so on.

The *has-a* relationship holds when the second concept is a component of the first but the two are not in any sense the same thing.

For example, a Car *has-a* Engine, although it is clear that it is not the case that Car *is-a* Engine or Engine *is-a* Car. Actually, a Car *is-a* Vehicle.

Inheritance of Code and Inheritance of Behavior

There are at least two different ways in which a concept can satisfy the *is-a* relationship with another concept.

Inheritance is the mechanism of choice when two concepts share a *structure* or *code* relationship with each other, while an interface is the more appropriate technique when two concepts share the *specification of behavior*, but no actual code.

In general, the class-subclass relationship should be used whenever a subclass can usefully inherit code, data values, or behavior from the parent class.

The interface mechanism should be used when the child class inherits only the specification of the expected behavior, but no actual code.

Composition and Inheritance Described

Two different techniques for software reuse after *composition* and *inheritance*.

One way to view these two different mechanism is a manifestation of the *has-a* rule and the *is-a* rule, respectively.

Although in most situations the distinction between *is-a* and *has-a* is clear-cut, however, sometimes it is difficult to determine which mechanism is most appropriate to use in a particular situation. One such situation is taken from the Java library and concerns the development of a Stack abstraction from an existing Vector data type.

A *stack* is an abstract data type that allows elements to be added or removed from one end only. The stack is specified by the following operations.

Operation	Action
empty	returns true if the given stack contains no elements, and returns false otherwise
push	returns the stack with a given element added to the top
peek	returns a copy of the element currently at the top of the given stack
pop	removes the top element from the given stack

Now we will see how we can implement the Stack using Vector with composition and inheritance.

Using Composition

Composition means implementing the class Stack in such a way that each instance of Stack contains a Vector object that holds the elements pushed on to the stack. The stack operations are then implemented in terms of the methods that are already defined for a vector object.

```
public class Stack {
        private Vector theData;

        public Stack()
        { theData = new Vector(); }

        public boolean empty()
        { return theData.isEmpty(); }

        public Object push (Object item)
        { theData.addElement (item); return item; }

        public Object peek()
        { return theData.lastElement(); }

        public Object pop() {
                Object result = theData.lastElement();
                theData.removeElementAt(theData.size()-1);
                return result;
        }
}
```

Note that because the Vector abstraction is stored as part of the data area for our stack, it must be initialized in the constructor of the Stack.

Composition makes no explicit or implicit claims about substitutability. When formed in this fashion, the data types Stack and Vector are entirely distinct and neither can be substituted in situations where the other is required.

Using Inheritance

Inheritance means extending the implementation of Vector to include methods implementing the operations that are required for a Stack.

```
public class Stack extends Vector {

        public Object push (Object item)
        { addElement (item); return item; }

        public Object peek()
        { return elementAt(size()-1); }
```

```
public Object pop() {
        Object obj = peek();
        removeElementAt(size()-1);
        return obj;
}
```

The most obvious features of this class in comparison to the earlier are the items that are missing. There are no local data fields. There is no constructor. Since no local data values need be initialized. The method isEmpty need not be provided, since the method is inherited already from the parent class Vector.

Comparing this method with the earlier version, both techniques are powerful mechanisms for code reuse, but unlike composition, inheritance carries an implicit assumption that subclass are, in fact, subtypes.

In Subsection 10.2.2, Budd says that using inheritance provides more useful functionality than the version using composition because, for example, the Vector method size automatically applies to Stack objects. While it is certainly true that size may be a useful operation on Stack objects, it represents an operation that is not part of the specification of the original abstract data type stack which means that the inheritance version of Stack is not a true implementation of the abstract data type stack. You may well think this is a trivial point but, as you will discover in later parts of the course, writing programs that do not conform precisely to the given specification can be a source of error and can make maintenance a nightmare. More problematic than the size method, this inherited version will allow a user to remove items from arbitrary positions in the stack. This is very far from the spirit of a stack specification, and a system developer who expects a component to be built from a stack would probably assume that items can only be taken off the top. This could easily lead to erroneous behavior.

Comparison between Composition and Inheritance

If the class to be built is a subtype of an existing class, it is clearly advantageous to extend the existing class (that is, use inheritance). This is because, in our view, the advantages here (in particular, to be able to use substitutability, and not to have to repeat the definition of common behavior) outweigh the disadvantages (to be able to see the state or data fields of the other class; that is, no encapsulation).

If, however, the proposed class cannot be considered to be a subtype of an existing class, that is, does not share the behavior of that class, but still needs the behavior of the other class in order to implement its own behavior, composition should be used to build it. Here, the advantages (in particular, encapsulating the data fields of the component class) outweigh the disadvantages (e.g. not being able to use substitutability, which will be spurious in any case, and not inheriting some operations that may be useful).

IMPORTANT: refer to Budd, pp. 172-174, for a detailed comparison.

Combining Inheritance and Composition

The Java input/output system (will be discussed later) provides an interesting illustration of combining both inheritance and composition.

To begin with, there is an abstract concept, and several concrete realizations of the concept.

For the file input system the abstract concept is the idea of reading a stream of bytes in sequence. This idea is embodied in the class InputStream, which defined a number of methods for reading byte values.

The concrete realization differs in the source of the data values. Values can come from an array of bytes being held in memory, from an external file, etc. There is a different subclass of InputStream for each of these, as shown in Figure 10.3 in Budd, pp. 175.

There is additional functionality that is sometimes required when using an input stream. Furthermore, this functionality is independent of the source for the byte values. One example is the ability to buffer input so as to have the possibility of rereading recently referenced bytes once again.

These features are provided by defining a subclass of InputStream, named FilterInputStream. Thus, using the principle of substitutability, a FilterInputStream can be used in places where an InputStream is expected. On the other hand, a FilterInputStream holds as a component another instance of InputStream, which is used as the source for data values.

The term *filter* is normally used in contexts where each item of a data stream is processed in the same way.

Thus, the class InputStream is both parent and component to FilterInputStream.

As requests for values are received, the FilterInputStream will access the InputStream it holds to get the values it needs, performing whatever additional actions are required (e.g. counting newline characters in order to keep track of the current line number).

```
public class FilterInputStream extends InputStream  {

        protected InputStream in;
        …
}
```

Because the component held by the FilterInputStream can be any type of InputStream, this additional functionality can be used to augment any type of InputStream, regardless of where the byte values originates.

IMPORTANT: Read page 5 and 6 in Section 4.1 of Block 1.

Block 1: Introduction to Java

Unit 4: Inheritance, Composition and Polymorphism

Section 2: Implications of Inheritance

In this section, you will study some of the programming implications of using inheritance for the purposes of polymorphism.

Objectives:

On completing this section you should be able to:

- describe what is meant by a polymorphic variable and relate it to the concept of inheritance;
- describe what is meant by, and exploit, the concepts of reference semantics and copy semantics as they affect assignment, tests for equality and copying objects.

Summary of Chapter 11

The Polymorphic Variable

A **polymorphic variable** is one that is declared as being of one type but may actually reference an object derived from another type.

In Java, a variable derived from a class `A` is potentially polymorphic if `A` has subclasses.

Polymorphic variables support substitutability.

An example of a polymorphic variable from the Pin Ball Game Construction Kit (Chapter 7) is a variable which was declared as holding a value of type `PinBallTarget`, when in a fact it would hold a `Hole` or a `ScorePad`.

Memory Layout

The majority of instructions carried out by a computer when executing a program involves accessing data or other instructions in memory. Each item of data or instruction has an address where it is located. This implies that the address must be known before the data or instruction can be accessed.

There are two fundamental occasions when an address can be calculated: compile time and execution time.

If an address can be computed at compile time, it minimizes the time required to access the data when the program is running. We often say that such an address is

static (it does not change at execution time) and *stack-based memory management* techniques are appropriate.

If an address is not known until the program is executed, that is, the actual address has to be computed at execution time, clearly it will take longer to access the data. Such an address is said to be **dynamic** and *heap memory management* has to be used. For the fastest execution it would be preferable if all addresses were static.

Unfortunately, this is not possible in general and, particularly, with polymorphic variables. Polymorphic variables can refer to objects that require different amounts of memory to hold them and it is not possible to know at compile time precisely which types of object will be referenced.

Assignment

Java uses **reference semantics** (sometimes known as pointer semantics) with regard to assignment, this means the following:

An assignment statement involving two variables, say `x=y`, causes the reference (address) in `x` to be changed to the same value as the reference in `y`. In other words, `x` and `y` are made to refer to the same object.

For another example, refer to the first notes page 5.

On the other hand, Java uses **copy semantics** for assignment of variables of primitive data type. Hence, variables of primitive data type store their values rather than references to these values. Consequently, after the assignment `y=x` (where both `x` and `y` are of type `int` and `x` holds the value `3`), `y` would then hold the value `3`; that is, a copy of the value held by `x`. This is known as assignment with **copy semantics**.

Clones

To obtain the effect of copy semantics with variables that reference objects, another operation (not assignment) is necessary.

One way to do this, is to explicitly create a new value, copying the internal contents from the existing value.

Example:

Suppose we create a simple class `Box` as follows:

```
public class Box {
        private int value;

        public Box() { value = 0; }
        public void setValue (int v) { value = v; }
        public int getValue() { return value; }
}
```

If making copies of Box is a common operation, we can provide a copying method in the original class as follows:

```
public class Box {
        .
        .
        .
        public Box Copy () { //make copy of box
                Box b = new Box();
                b.setValue(getValue());
                return b;
        }
        .
        .
        .
}
```

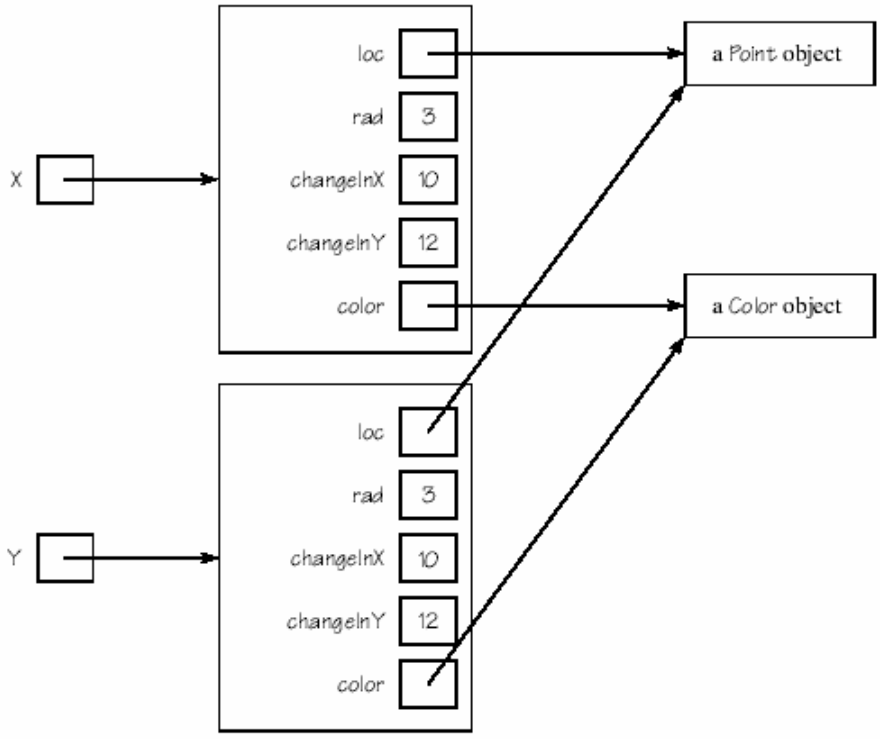
A copy of a box can be created by invoking the copy() method as follows:

```
// create new box with same value as x
Box y = x.copy();
```

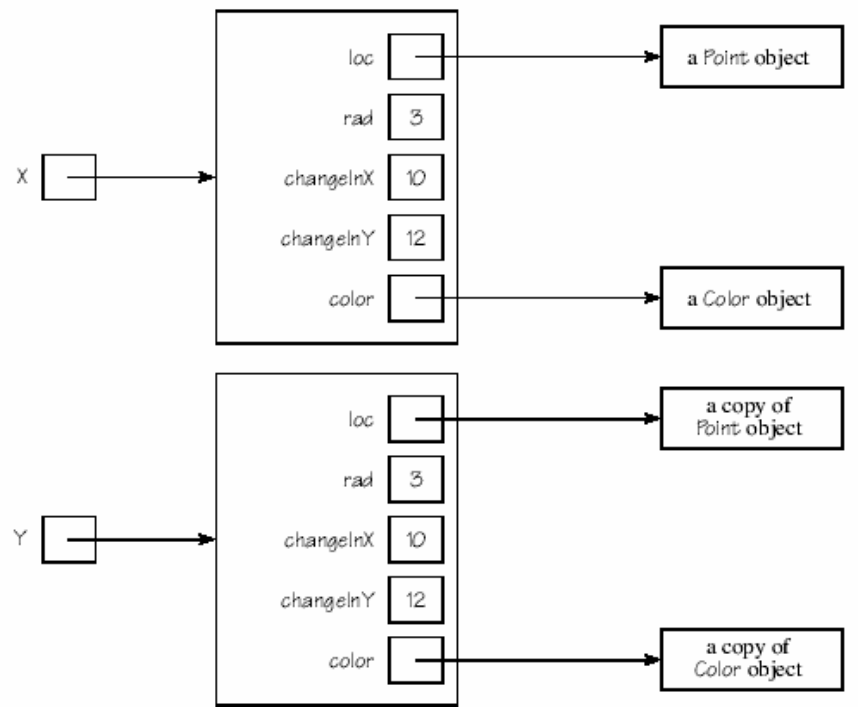
IMPORTANT: Read Budd pp. 187-189

There two types of copy:

Shallow copy where the copy results in two distinct objects but ones that share a common value. This is shown in the figure below.



Deep copy results in two objects that not only themselves distinct, but that point to values that are also distinct. This is shown in the figure below.



Parameters as a Form of Assignment

Note that passing a variable as an argument to a member function can be considered to be a form of assignment, in that parameter passing, like assignment, results in the same value being accessible via two different names.

Example:

Consider the method sneaky in the following example, which modifies the value held in a box that is passed through a parameter value.

```
public class BoxTest {
        public static void main (String [] args) {
                Box x = new Box();
                x.setValue(7);
                sneaky(x);
                System.out.println("contents of x "+ x.getValue());
        }
        static void sneaky(Box y) {
                y.setValue(11); //change the value of parameter
        }
}
```

The output of the above code will be:

Contents of x 11

Equality Test

The == operator

For primitive data types we use the == operator to test quality as in the following expressions (both will return true):

```
7 == (3 + 4)
2 == 2.0
```

For objects the == operator tests for equality of object identity (memory locations); that is, whether the two variables being compared refer to the same object.

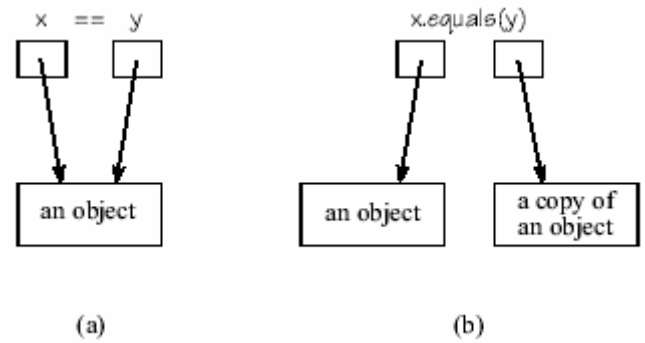
The Java compiler does apply type checking rules to the two arguments, which will help detect many programming errors. For example, a numeric value can be compared to another numeric. A numeric cannot be compared to different type of object (e.g. a String). Two objects can be compared if they are the same type, or if the class of one can be converted into the class of the second. For example, a variable that was declared to be an instance of class Shape could be compared with a variable of type Circle, since a Circle would be converted to a Shape.

Any object can be compared to the constant null, since the value null can be assigned to any object type.

The equals() method

The equals() method either inherited from Object class or overridden, tests whether two separate objects are copies of each other, that is, have identical values in their data fields (not pointing to the same object).

The following figure illustrates the difference between == and equals(), where equals has been redefined for the objects in question. Figure (a) shows the only situation in which == is certain to be true, and Figure (b) shows a situation in which equals() may be true (because the values of the fields in the two objects are the same (and == is definitely false).



The equals() method can be added to the Box class as follows:

```
public class Box {

   private int value;
   public Box ();

   public void setValue ();

   public int getValue ();

   public boolean equals (Object arg) {
            // if arg references a Box object
      if (arg instanceof Box) {
            // assign it to a new variable
         Box argBox = (Box) arg;
            // check whether 'value' of current box equals the 'value' of input box
         if (this.value == argBox.value) {
            return true;
         }
      }
      return false;
   }
}
```

A good rule of thumb is to use == when testing numeric quantities, and when testing an object against the constant null. In other situations the method equals() should be used.

Garbage Collection

The garbage collector in Java recycles memory resources by periodically recovering the memory being used by objects on the heap that are no longer referenced by any variable in the program.

A major advantage of having a garbage collector is that it saves the programmer having to write code that will do the task. Leaving garbage collection to the programmer can be a source of error because an object removed by the programmer might still have other references of which the programmer is unaware.

A major disadvantage is the degradation in performance caused by having to interrupt program execution while the garbage collector searches the memory looking for and removing unreferenced objects. However, the improvement in reliability is well worth the time overhead.

Block 1: Introduction to Java

Unit 4: Inheritance, Composition and Polymorphism

Section 3: Polymorphism

Polymorphism, which means appearing in many forms, is a natural result of inheritance and occurs in a variety of guises. In this section you will study the various types of polymorphism.

Objectives:

On completing this section you should be able to:

- recognize pure polymorphism;
- recognize overloading of method names;
- describe the concept of coercion;
- distinguish between overloading of method names in classes unrelated by inheritance and overloading of method names within the same class definition;
- recognize parametric overloading;
- recognize overriding and distinguish between replacement and refinement semantics;
- understand the use of abstract methods;
- appreciate why polymorphism may not be an efficient mechanism but provides other characteristics that are more useful to the programmer such as ease of development and use, and readability of code.

Summary of Chapter 12

The term polymorphic means roughly "many forms".

Varieties of Polymorphism

In OO languages, polymorphism is a natural result of the *is-a* relationship and of mechanisms of message passing, inheritance, and the concept of substitutability.

Polymorphism is an umbrella term used to describe a variety of different mechanism found in programming languages. In the following we will discuss different forms of polymorphism.

Polymorphic Variables

A *polymorphic variable* is one that can hold, or refer to, values of different types (i.e. subtypes of the type of the variable).

Polymorphic variables embody the principle of substitutability. In other words, although there is an expected type for any variable, the actual type can be from any value that is a subtype of the expected type.

In a dynamically typed language (e.g. Smalltalk), *all* variables are potentially polymorphic since any variable can hold (or refer to) values of any type.

In a statically typed language (e.g. Java), polymorphism is restricted to a variable being able to refer to objects of its class (i.e. its *static class*) and subclasses (i.e. its *dynamic class*).

Overloading

We say a method name is overloaded if two or more function bodies are associated with it. Note that overloading is a necessary part of overriding but the two terms are not identical and overloading can occur without overriding.

In overloading, it is the method name that is polymorphic—it has many forms.

What does it mean to say that a method name is overloaded?

There are the following three forms for overloading:

- where the method heading is the same as that in the parent class, but the body is redefined (this is the case of overriding);
- where methods in the same class (or classes related by inheritance) have the same names but different arguments;
- where methods in two or more classes not linked by inheritance have the same name (whether or not they have the same arguments is not an issue) (e.g. the `isEmpty()` method defined by the classes `Vector`, `HashTable` and `Rectangle` – this method is used to determine if an object is empty, however, the exact meaning of empty will differ depending upon the circumstances).

Note that there is nothing intrinsic to overloading that requires the methods associated with an overloaded name to have any semantic similarity. In other words, *overloading does not imply similarity*.

Parametric overloading occurs when there are two or more methods in the same class definition with the same name but that differ in either (or both) the number of arguments or the type of arguments. Parametric overloading is most often found in constructor methods.

Overloading is a necessary prerequisite to other forms of polymorphism we will consider: overriding, deferred methods and pure polymorphism.

IMPORTANT: Read Budd pp. 201-202

Overriding

Overriding occurs when a method is defined in a class and a new method with the same name is defined in a subclass so that access to the method in the superclass is hidden. *Overloading* occurs when an identifier denotes more than one object (or method or operator). When a method is overridden, its name also refers to another method and hence **overriding leads to overloading**.

There are two types of overriding:

Overriding by replacement (referred to as *replacement semantics*) occurs when the code of the method in the parent class is not executed (in effect, the code of the method in the superclass has been replaced by the code of the method in the subclass).

Overriding by refinement (referred to as *refinement semantics*) occurs when the code of subclass invokes the code of the superclass.

Normally, overridden methods use replacement semantics. Constructors always use refinement semantics. A constructor for a child class will always invoke the constructor for the parent class. This invocation will take place before the code for the constructor is executed. If the constructor for the parent class requires arguments, the pseudovariable super is used as if it were a method. For example:

```
Class DeckPile extends CardPile {
        DeckPile (int x, int y) {
                // first initialize parent
                super(x, y);
                // do other stuff
                .
                .
                .
        }
}
```

When used in this fashion, the call on the parent constructor must be the first statement executed.

If no call on super is made explicitly, an implicit call will be made to the constructor in the parent class that takes no arguments (e.g. a default constructor).

Abstract (or deferred) Methods

An **abstract method** (also known as a **deferred method**) is a method that is specified in the parent class, that is, it is without a body, but must be implemented (i.e. the body must be defined) in a descendant class (e.g. the child class or a descendant of the child class).

Interfaces contain abstract methods only. It can be considered to be a generalization of overriding where the behavior described in a parent class is modified by the child class.

There are two advantages of abstract methods:

1. Conceptual, in that their use allows the programmer to think of an activity as associated with an abstraction at a higher level than may actually be the case (for examples read Budd, pp. 205-206).
2. In statically typed OO languages (e.g. Java) a programmer is permitted to send a message to an object only if the compiler can determine that there is in fact a

corresponding method that matches the message selector (for examples read Budd, pp. 206).

Pure Polymorphism

The term **pure polymorphism** refers to situations where a method can be used with arguments of different type. This means that the formal arguments act like polymorphic variables (that is, the actual arguments can be subtypes of the formal argument). This means that the formal arguments (which are local variables) are polymorphic. That is, the actual arguments can be of different types, as long as they are subtypes of the type of the formal arguments.

Efficiency and Polymorphism

A programmer should not be overly concerned with the loss of efficiency due to the use of polymorphic programming techniques because they offer the possibility of rapid program development, consistent application behavior, and code reuse.

Exercise 3.1 (Budd, Chapter 12, Exercise 1)

Describe the various types of polymorphism found in the PinBall Game application presented in Chapter 7.

Solution

The PinBallGame program contains examples of all types of polymorphism. Here are some examples of each type.

Pure polymorphism

In the class PinBallGame, the variable targets is a polymorphic variable. The method addElement can take arguments of different type.

Overloading

The name hitBy is used for a method in class Wall and in class Hole (see Budd, Figures 7.7 and 7.8).

Overriding

The class Peg overrides the paint method (inherited from ScorePad).

The class MouseKeeper overrides the methods mousePressed and mouseReleased (inherited from MouseAdapter).

Deferred methods

The interface PinBallTarget specifies four deferred methods (intersects, moveTo, paint and hitBy).