Chapter 5: Enhancing Classes

# Enhancing Classes

We can now explore various aspects of classes and objects in more detail.

Chapter 5 focuses on:
- object references and aliases
- passing objects as parameters
- the static modifier
- nested classes
- interfaces and polymorphism
- events and listeners
- animation

## References

Recall from Chapter 2 that an object reference holds the memory address of an object.

Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object.

```java
ChessPiece bishop1 = new ChessPiece();
```

## Assignment Revisited

The act of assignment takes a copy of a value and stores it in a variable.

For primitive types:

```java
num2 = num1;
```

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>num1</td>
<td>num2</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>num1</td>
<td>num2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

## Reference Assignment

For object references, assignment copies the memory location:

```java
bishop2 = bishop1;
```

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>bishop1</td>
<td>bishop2</td>
</tr>
<tr>
<td>bishop1</td>
<td>bishop2</td>
</tr>
<tr>
<td>bishop1</td>
<td>bishop2</td>
</tr>
<tr>
<td>bishop1</td>
<td>bishop2</td>
</tr>
</tbody>
</table>

## Aliases

Two or more references that refer to the same object are called aliases of each other.

One object (and its data) can be accessed using different variables.

Aliases can be useful, but should be managed carefully.

Changing the object’s state (its variables) through one reference changes it for all of its aliases.
**Garbage Collection**

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- It is useless, and therefore called garbage
- Java performs automatic garbage collection periodically, returning an object's memory to the system for future use
- In other languages, the programmer has the responsibility for performing garbage collection

**Passing Objects to Methods**

- Parameters in a Java method are passed by value
- This means that a copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Passing parameters is essentially an assignment
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

**Passing Objects to Methods**

- What you do to a parameter inside a method may or may not have a permanent effect (outside the method)
- See ParameterPassing.java (page 226)
- See ParameterTester.java (page 228)
- See Num.java (page 230)
- Note the difference between changing the reference and changing the object that the reference points to

**The static Modifier**

- In Chapter 2 we discussed static methods (also called class methods) that can be invoked through the class name rather than through a particular object
- For example, the methods of the Math class are static
- To make a method static, we apply the static modifier to the method definition
- The static modifier can be applied to variables as well
- It associates a variable or method with the class rather than an object

**Static Methods**

```java
class Helper
{
    public static int triple (int num)
    {
        int result;
        result = num * 3;
        return result;
    }
}
```

Because it is static, the method could be invoked as:

```java
value = Helper.triple (5);
```
Static Variables

- Static variables are sometimes called class variables
- Normally, each object has its own data space
- If a variable is declared as static, only one copy of the variable exists
  
  ```java
  private static float price;
  ```
- Memory space for a static variable is created as soon as the class in which it is declared is loaded

- All objects created from the class share access to the static variable
- Changing the value of a static variable in one object changes it for all others
- Static methods and variables often work together
  
  See `CountInstances.java` (page 233)
  
  See `MyClass.java` (page 234)

Nested Classes

- In addition to a class containing data and methods, it can also contain other classes
- A class declared within another class is called a nested class

- A nested class has access to the variables and methods of the outer class, even if they are declared private
  
  In certain situations this makes the implementation of the classes easier because they can easily share information
  
  Furthermore, the nested class can be protected by the outer class from external use
  
  This is a special relationship and should be used with care

- A nested class produces a separate bytecode file
  
  If a nested class called Inside is declared in an outer class called Outside, two bytecode files will be produced:

  ```
  Outside.class
  Outside$Inside.class
  ```
  
  Nested classes can be declared as static, in which case they cannot refer to instance variables or methods
  
- A nonstatic nested class is called an inner class

Interfaces

- A Java interface is a collection of abstract methods and constants
  
  An abstract method is a method header without a method body
  
  An abstract method can be declared using the modifier abstract, but because all methods in an interface are abstract, it is usually left off
  
  An interface is used to formally define a set of methods that a class will implement
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2(float value, char ch);
    public boolean doTheOther(int num);
}

A semicolon immediately follows each method header.

public class CanDo implements Doable
{
    public void doThis()
    {
        // whatever
    }
    public void doThat()
    {
        // whatever
    }
    // etc.
}

Each method listed in Doable is given a definition.

Polymorphism via Interfaces

+ An interface name can be used as the type of an object reference variable

    Doable obj;

+ The obj reference can be used to point to any object of any class that implements the Doable interface

+ The version of doThis that the following line invokes depends on the type of object that obj is referring to:

    obj.doThis();
Interfaces

- The Java standard class library contains many interfaces that are helpful in certain situations.
- The `Comparable` interface contains an abstract method called `compareTo`, which is used to compare to objects.
- The `String` class implements `Comparable` which gives us the ability to put strings in alphabetical order.
- The `Iterator` interface contains methods that allow the user to move through a collection of objects easily.

Events

- An event is an object that represents some activity to which we may want to respond.
- For example, we may want our program to perform some action when the following occurs:
  - the mouse is moved
  - a mouse button is clicked
  - a graphical button is clicked
  - a keyboard key is pressed
  - a timer expires
- Often events correspond to user actions, but not always.

Events and Listeners

- The Java standard class library contains several classes that represent typical events.
- Certain objects, such as an applet or a graphical button, generate (fire) an event when it occurs.
- Other objects, called a listeners, respond to events.
- We can write listener objects to do whatever we want when an event occurs.

Listener Interfaces

- We can create a listener object by writing a class that implements a particular listener interface.
- The Java standard class library contains several interfaces that correspond to particular event categories.
- For example, the `MouseListener` interface contains methods that correspond to mouse events.
- After creating the listener, we add the listener to the component that might generate the event to set up a formal relationship between the generator and listener.

Mouse Events

- The following are mouse events:
  - `mousePressed` - the mouse button is pressed down
  - `mouseReleased` - the mouse button is released
  - `mouseClicked` - the mouse button is pressed and released
  - `mouseEntered` - the mouse pointer is moved over a particular component
  - `mouseExited` - the mouse pointer is moved off of a particular component
- Any given program can listen for some, none, or all of these.
  - See `Dots.java` (page 246)
  - See `DotsMouseListener.java` (page 248)
### Mouse Motion Events

- The following are called *mouse motion events*:
  - *mouse moved* - the mouse is moved
  - *mouse dragged* - the mouse is moved while the mouse button is held down
- There is a corresponding *MouseMotionListener* interface
- One class can serve as both a generator and a listener
- One class can serve as a listener for multiple event types
- See *RubberLines.java* (page 249)

### Key Events

- The following are called *key events*:
  - *key pressed* - a keyboard key is pressed down
  - *key released* - a keyboard key is released
  - *key typed* - a keyboard key is pressed and released
- The *KeyListener* interface handles key events
- Listener classes are often implemented as inner classes, nested within the component that they are listening to
- See *Direction.java* (page 253)

### Animations

- An animation is a constantly changing series of pictures or images that create the illusion of movement
- We can create animations in Java by changing a picture slightly over time
- The speed of a Java animation is usually controlled by a *Timer* object
- The *Timer* class is defined in the *javax.swing* package

### Animations

- A *Timer* object generates and *ActionEvent* every n milliseconds (where n is set by the object creator)
- The *ActionListener* interface contains an *actionPerformed* method
- Whenever the timer expires (generating an *ActionEvent*) the animation can be updated
- See *Rebound.java* (page 258)