Chapter 5: Enhancing Classes

Now we can explore various aspects of classes and objects in more detail.

Chapter 5 focuses on:
- object references and aliases
- passing objects references as parameters
- the static modifier
- exceptions
- interfaces
- nested classes and inner classes
- dialog boxes
- GUI components, events, and listeners

References

- Recall from Chapter 2 that an object reference variable 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();
```

The null Reference

- An object reference variable that does not currently point to an object is called a null reference.
- The reserved word null can be used to explicitly set a null reference:

```java
name = null;
```

- or to check to see if a reference is currently null:

```java
if (name == null)
    System.out.println("Invalid");
```
The null Reference

- An object reference variable declared at the class level (an instance variable) is automatically initialized to null
- The programmer must carefully ensure that an object reference variable refers to a valid object before it is used
- Attempting to follow a null reference causes a NullPointerException to be thrown
- Usually a compiler will check to see if a local variable is being used without being initialized

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>5</td>
</tr>
<tr>
<td>num2</td>
<td>12</td>
</tr>
<tr>
<td>num1</td>
<td>5</td>
</tr>
<tr>
<td>num2</td>
<td>5</td>
</tr>
</tbody>
</table>

The this Reference

- The this reference allows an object to refer to itself
- That is, the this reference, used inside a method, refers to the object through which the method is being executed
- Suppose the this reference is used in a method called tryMe
- If tryMe is invoked as follows, the this reference refers to obj1:
  ```java
  obj1.tryMe();
  ```
- But in this case, the this reference refers to obj2:
  ```java
  obj2.tryMe();
  ```

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>bishop2</td>
<td>bishop1</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 reference 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.

Objects as Parameters

- 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 therefore similar to an assignment statement.
  - When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other.

Testing Objects for Equality

- The `==` operator compares object references for equality, returning `true` if the references are aliases of each other.
  ```java
  bishop1 == bishop2
  ```
- A method called `equals` is defined for all objects, but unless we redefine it when we write a class, it has the same semantics as the `==` operator.
  ```java
  bishop1.equals(bishop2)
  ```
- We can redefine the `equals` method to return `true` under whatever conditions we think are appropriate.

Passing Objects to Methods

- What you do with a parameter inside a method may or may not have a permanent effect (outside the method).
- See `ParameterPassing.java` (page 269).
- See `ParameterTester.java` (page 270).
- See `Num.java` (page 266).
- 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:
  ```java
  Math.sqrt (25)
  ```
- To write a static method, 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 with an object.

Static Variables

- Static variables are also called class variables.
- Normally, each object has its own data space, but 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 when the class in which it is declared is loaded.
- All objects created from the class share static variables.
- The most common use of static variables is for constants.

Static Methods

- 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:
  ```java
  Math.sqrt (25)
  ```
- To write a static method, 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 with an object.

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

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

```java
value = Helper.triple (5);
```
The static Modifier

- Static methods and static variables often work together
- See CountInstances.java (page 273)
- See Slogan.java (page 275)

Exception Handling

- Java has a predefined set of exceptions and errors that can occur during execution
- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it at another place in the program

Exceptions

- An exception is an object that describes an unusual or erroneous situation
- Exceptions are thrown by a program, and may be caught and handled by another part of the program
- A program can be separated into a normal execution flow and an exception execution flow
- An error is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught

Exception Handling

- If an exception is ignored by the program, the program will terminate abnormally and produce an appropriate message
- The message includes a call stack trace that indicates the line on which the exception occurred
- The call stack trace also shows the method call trail that lead to the attempted execution of the offending line
- See Zero.java (page 277)
The throw Statement

- Exceptions are thrown using the `throw` statement.
- Usually a throw statement is nested inside an if statement that evaluates the condition to see if the exception should be thrown.
- The following statement throws a `NoSuchElementException`:
  
  ```java
  throw new NoSuchElementException();
  ```

- See *Throwing.java* (page 278)

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, usually it is left off.
- An interface is used to establish, as a formal contract, a set of methods that a class will implement.

- An interface cannot be instantiated.
- Methods in an interface have public visibility by default.
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for each abstract method in the interface.
- If a class asserts that it implements an interface, it must define all methods in the interface.

```java
public interface Doable {
    public void doThis();
    public int doThat();
    public void doThis2 (double value, char ch);
    public boolean doTheOther (int num);
}
```
Interfaces

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

Each method listed in `Doable` is given a definition.

- A class that implements an interface can implement other methods as well
- See `Complexity.java` (page 279)
- See `Question.java` (page 281)
- See `MiniQuiz.java` (page 282)
- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants

Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the implements clause
- The class must implement all methods in all interfaces listed in the header

```java
class ManyThings implements interface1, interface2 {
    // all methods of both interfaces
}
```

- The Java standard class library contains many helpful interfaces
- The `Comparable` interface contains an abstract method called `compareTo`, which is used to compare two objects
- The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order
- The `List` interface is implemented by classes that represent an ordered collection of elements
- The `Iterator` interface contains methods that allow the user to move easily through a collection of objects
The Comparable Interface

- The Comparable interface provides a common mechanism for comparing one object to another
  
  ```java
  if (obj1.compareTo(obj2) < 0)
      System.out.println("obj1 is less than obj2");
  ```

- The result is negative if obj1 is less than obj2, 0 if they are equal, and positive if obj1 is greater than obj2

- When a programmer writes a class that implements the Comparable interface, it should follow this intent

- It's up to the programmer to determine what makes one object less than another

The List Interface

- The List interface represents an ordered collection of elements

- The size method returns the number of elements in the list

- The add method adds an element to the list

- The iterator and listIterator methods return iterators of the elements in the list

Iterator and ListIterator Interfaces

- The Iterator and ListIterator interfaces provide a means of moving through a collection of objects, one at a time

- The hasNext method returns a boolean result (true if there are items left to process)

- The next method returns the next object in the iteration

- The remove method removes the object most recently returned by the next method

- The ListIterator interface has additional methods (add and set) that insert or replace an element in the list

Identifying Classes and Objects

- During the design stage, classes and objects need to be identified

- As a start, examine the program requirements

- Objects are generally nouns

- A class represents a group of objects with similar behavior

- For example, to represent products, we may need a class called Product

- Strike a good balance between classes that are too general and those that are too specific
Designing Classes

- When designing a class, there are two pieces of information to think about:
  - State (how an object is represented)
  - Behavior (what an object does)
- The state becomes the instance variables of an object
- The behavior becomes the methods
- When thinking about behavior, you should think about how others might want to use the object

Nested Classes

- In addition to containing data and methods, a class can 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 enclosing class, even if they are declared private
- In certain situations this makes the implementation of the classes easier because they can share information easily
- Furthermore, the nested class can be protected by the enclosing 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 are produced:
  ```
  Outside.class
  Outside$Inside.class
  ```
- Nested classes can be declared as static, in which case they cannot refer to instance variables or methods
Inner Classes

- A nonstatic nested class is called an *inner class*
- An inner class is associated with each instance of the enclosing class
- An instance of an inner class can exist only within an instance of an enclosing class

Dialog Boxes

- A *dialog box* is a graphical window that pops up on top of any currently active window for the user
- The Swing API contains a class called `JOptionPane` that simplifies the creation and use of basic dialog boxes
- There are three categories of `JOptionPane` dialog boxes
  - A *message dialog* displays an output string
  - An *input dialog* presents a prompt and a single input text field
  - A *confirm dialog* presents the user with a simple yes-or-no question
- See `EvenOdd.java` (page 294)

The EvenOdd Program

Graphical User Interfaces

- A Graphical User Interface (GUI) is created with at least three kinds of objects
  - components
  - events
  - listeners
- A GUI *component* defines a screen element to display information or allow the user to interact with the program
  - buttons, text fields, labels, menus, etc.
- A *container* is a special component that holds and organizes other components
  - dialog boxes, applets, frames, panels, etc.
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
  - the mouse is dragged
  - a graphical button is clicked
  - a keyboard key is pressed
  - a timer expires
- Events often correspond to user actions, but not always.

Events and Listeners

- The Java standard class library contains several classes that represent typical events.
- Components, such as an applet or a graphical button, generate (fire) an event when it occurs.
- Other objects, called listeners, wait for events to occur.
- We can write listener objects to do whatever we want when an event occurs.
- A listener object is often defined using an inner class.

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.
Creating GUIs

- To create a program with a GUI:
  - define and set up the components
  - create listener objects
  - set up the relationships between the listeners and the components which generate events of interest
  - define what happens in response to each event

- A push button is a component that allows the user to initiate an action with the press of the mouse button
  - defined by the JButton class
  - generates an action event

- A label is a component that displays a line of text (or an image, or both)
  - defined by the JLabel class

The PushCounter Program

- The init method of an applet can be used to set up the GUI and add each component to the applet container
- The Swing version of the Applet class is called JApplet
- In a JApplet, components are added to the applet's content pane
- The content pane is retrieved using the getContentPane method
- A JButton generates an action event
- See PushCounter.java (page 297)

Action Listeners

- The interface corresponding to an action event is called ActionListener, which defines only one method, called actionPerformed
- The ButtonListener inner class implements the ActionListener interface in the PushButton program
- When the button is pushed, the JButton object invokes the actionPerformed method, passing it an ActionEvent
- The listener method may or may not make use of the event object passed to it
GUI Applications

- A frame is a container component used for stand-alone GUI-based applications
- A panel is a container, but unlike a frame, it cannot be displayed on its own
  - it must be added to another container
  - it helps organize the components in a GUI
- See Fahrenheit.java (page 300)
- See FahrenheitGUI.java (page 302)

The Fahrenheit Program

Summary

- Chapter 5 has focused on:
  - object references and aliases
  - passing objects references as parameters
  - the static modifier
  - exceptions
  - interfaces
  - nested classes and inner classes
  - dialog boxes
  - GUI components, events, and listeners