CSI1102: Introduction to Software Design

Chapter 5:
Enhancing Classes

Learning objectives: Enhancing Classes

Understand what the following entails

- Different object references and aliases
- Passing objects (references) as parameters
- The static modifier: static variables and methods
- Wrapper classes for primitive data types
- Nested and inner classes
- Interfaces for software design
- GUI components, dialog boxes, events, and listeners

More about 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


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


The null Reference

- An object reference variable that does not currently point to an object is called a *null reference*
- An attempt to follow a null reference causes a `NullPointerException` to be thrown
- For example

```
String name;
```

declares an object reference variable, but does not create a `String` object for it to refer to.

- Therefore, the variable `name` contains a null reference

The this Reference

- The `this` reference allows an object to refer to itself.
- Inside a method, the `this` reference can be used to refer to the currently executing object
- For example,

```
if(this.position == piece2.position)
     result = false;
```

clarifies which position is being referenced

- The `this` reference refers to the object through which the method containing the code was invoked

The this reference

- The this reference can also distinguish the parameters of a constructor from the corresponding instance variables with the same names
- Avoid: Not very readable!!!!

```
Public Account (Sring name, long acctNumber,
                double balance)
 {
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
 }
```

Assignments Revisited

- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types: `num2 = num1;`

Before

num1	num2
5	12

After

num1	num2
5	5

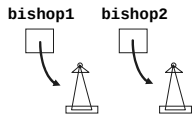
7

Object Reference Assignment

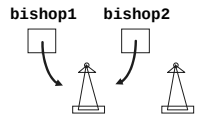
- For object references, assignment copies the memory location:

```
bishop2 = bishop1;
```

Before



After



8

What are 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

9

Testing Objects for Equality

- The `==` operator compares object references for equality, returning `true` if the references are aliases of each other
- 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

```
bishop1.equals(bishop2);
```

returns true if both references refer to the same object

- We can redefine the `equals` method to return `true` under whatever conditions we think are appropriate

10

Garbage Collection in Java

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

11

Passing 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 essentially **like an assignment statement**
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

12

Passing Objects to Methods

- What you do using a parameter inside a method may or may not have a permanent effect (outside the method)
- ParameterPassing.java (page 277)
- ParameterTester.java (page 279)
- Num.java (page 281)
- Note the difference between changing the reference and changing the object that the reference points to

ParameterPassing.java

```
public class ParameterPassing
    {
     // sets up 3 variables and illustrate parameter passing
     public static void main (String [] args)
    {
          ParameterTester tester = new ParameterTester();

          int a1 = 111;
          Num a2 = new Num (222);
          Num a3 = new Num (333);

          System.out.println("Before ChangeValues: ");
          System.out.println("a1  a2  a3: " + a1 + " " + a2 + " " + a3);

          tester.changeValues(a1, a2, a3);

          System.out.println("After ChangeValues: ");
          System.out.println("a1  a2  a3: " + a1 + " " + a2 + " " + a3);
    }
}
```


ParameterTester.java

```
public class ParameterTester
    {
     public void changeValues(int f1, Num f2, Num f3)
    {
          System.out.println("Before changing the values: ");
          System.out.println("f1  f2  f3: " + f1 + " " + f2 + " " + f3);

          f1 = 999;
          f2.setValue(888);
          f3 = new Num(777);

          System.out.println("After changing the values: ");
          System.out.println("f1  f2  f3: " + f1 + " " + f2 + " " + f3);
    }
}
```


Num.java

```
public class Num
{
    private int value;

    // Constructor
    public Num (int update)
    {
          value = update;
    }

    // set up a value
    public void setValue(int update)
    {
          value = update;
    }

    // toString
    public String toString()
    {
          return value + " ";
    } }
```


The results: Parameter passing

Before ChangeValues:
a1 a2 a3: 111 222 333

Before changing the values:
f1 f2 f3: 111 222 333

After changing the values:
f1 f2 f3: 999 888 777

After ChangeValues:
a1 a2 a3: 111 888 333

What are Static Variables?

- Associated **with the class** rather than **with an object**
- Static variables sometimes are 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**

```
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 access to the static variable →
  - **Changing the value of a static variable in one object changes it for all others**

More about Static Methods

- Static methods (also called class methods) 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
- Static methods **cannot reference instance variables**, because instance variables don't exist until an object exists

```
public static int abs(int num);
```

19

More about Static Methods: An example

```
class Helper

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

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

```
value = Helper.triple (5);
```

20

Static Methods and Variables: CountInstances.java

```
public class CountInstances
{
        public static void main ( String[] args)
        {
                Slogan obj;

                obj = new Slogan("Hello world");
                obj = new Slogan("Talk is cheap.");
                obj = new Slogan("Don't worry, be happy.");

                System.out.println("Slogans created: " + Slogan.getCount());
        }
}
```

21

Static Methods and Variables: Slogan.java

```
public class Slogan
{
        private String phrase;
        public static int count = 0;

        // the constructor
        public Slogan (String str)
        {
                phrase = str;
                count++;
        }

        // returns the number of objects of this class created
        public  static int getCount()
        {
                return count;
        }
}
```

22

What are Wrapper Classes?

- A wrapper class represents a particular primitive type
- For example

```
Integer ageObj = new Integer (20);
```

  uses the `Integer` class to create an object which effectively represents the integer 20 as an object
- Why do we need this?
  - This is useful when a program **requires an object** instead of a primitive type
- There is a wrapper class in the `Java.lang` package for each primitive type, see Figure 5.4

Primitive type (PT)	Wrapper Class
byte	Byte
int	Integer
char	Character, etc. for other PTs

23

Some methods of the Integer class: See p.287

`Integer (int value)` // constructor: create a new Integer object

```
byte byteValue ()
double doubleValue ()
```
// Return the value of this integer as the corresponding PT

```
static int parseInt (String str)
```
   // returns the int corresponding to the value stores in the string

```
static String toBinaryString (int num)
```
   // returns a string representation of the integer in the corresponding base

24

Java I/O

- Java I/O is accomplished using **objects** that represent **streams of data**
  - A *stream* is an ordered sequence of bytes
- The `System.out` object represents a standard *output stream*, which defaults to the monitor screen
- Reading keyboard input is more complicated… **more about it in Chapter 8**

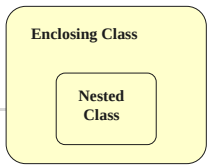
Keyboard Input Revisited

- Input can be read from the keyboard without using the `Keyboard` class and Wrapper Classes
- See Wages2.java (page 289): **More in Chapter 8**

```
import java.io.*;
…
BufferedReader in =
new BufferedReader(new InputStreamReader(System.in));
…
String name = in.readLine();

int hours = Integer.parseInt(in.readLine());
```


Nested Classes



- In addition to containing data and methods, a class can contain another *nested class*
- A nested class has access to the variables and methods of the enclosing class, even if they are declared private
- This is a special relationship and should be used with care

Nested Classes

- 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**

Inner Classes: A Non-static nested class

- An inner class is associated with each instance of the enclosing class & can exist only within an instance of an enclosing class

```
public class TestInner
  {
      // create and manipulate an outer object
      public static void main (String[] args)
      {
            Outer out = new Outer();

            System.out.println(out);
            out.changeMessage();
            System.out.println(out);
      }
  }
```


Nested Classes: Outer.java

```
public class Outer
  {
      private int num;
      private Inner in1, in2;

      public Outer()
      {
            num = 2365;
            in1 = new Inner ("Hello");
            in2 = new Inner ("Hello again");
      }

      public void changeMessage()
      {
            in1.message = "Eat desert first";
            in2.message = "Another miracle";
      }

      public String toString()
      {
            return in1 + "\n" + in2;
      }
```

Continued…

Nested classes: Outer.java

```
    // The inner class
    private class Inner
    {
          public String message;

          public Inner (String str)
          {
                message = str;
          }

          public String toString()
          {
                num++;
                return message + "\nOuter number = " + num;
          }

    }
}
```


Nested classes: The output

Hello
Outer number = 2366
Hello again
Outer number = 2367
Eat desert first
Outer number = 2368
Another miracle
Outer number = 2369

Interfaces: Useful for software design

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
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

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 **define a set of methods formally** that a class will implement

Interfaces: An example

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

More about Interfaces

- 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 or the compiler will produce errors.
- 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 that implements an interface can implement other methods as well
- A class can implement multiple interfaces

```
class ManyThings implements interface 1,
interface2, interface3
{
    //    all methods of all interfaces
}
```

- The interfaces are listed in the implements clause, separated by commas
- The class must implement all methods in all interfaces listed in the header

37

More about 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` which gives us the ability to put strings in alphabetical order
- The `Iterator` interface contains methods that allow the user to move easily through a collection of objects

38

The Comparable Interface

- The `Comparable` interface provides a common mechanism for comparing one object to another

```
if (obj1.compareTo(obj2) < 0)
    System.out.println ("obj1 is less than obj2");
```

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

39

The Iterator Interface

- The `Iterator` interface provides 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 an object
- The `remove` method removes the object most recently returned by the `next` method

40

About GUIs and Dialog Boxes (**Much more in Chapter 9**)

- 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 to allow the user to interact with the program, including push buttons, text fields, etc.
- The Swing package contains a class called `JOptionPane` that simplifies the creation and use of basic **dialog boxes**

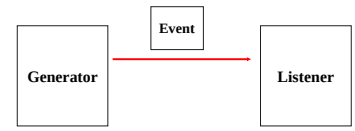
41

What is an Dialog Box?

- A graphical component used to interact with the user,
  - 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

42

About Events and Listeners



This object may generate an event

This object waits for and responds to an event

When an event occurs, the generator calls the appropriate method of the listener, passing an object that describes the event

Listener Interfaces

- We can create a listener object by writing a class that implements a particular *listener interface*
- 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
- See PushCounter.java on p.305

An event listener: PushCounter.java (an extract)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PushCounter extends JApplet
{
   private int pushes;
   …
   public void init ()
   {
        pushes = 0;
        push = new JButton("PUSH ME!");
        push.AddActionListener(new ButtonListener());
   …
   }
```

Summary: Enhancing Classes

Understand what the following entails

- Different object references and aliases
- Passing objects (references) as parameters
- The static modifier: static variables and methods
- Wrapper classes for primitive data types
- Nested and inner classes
- Interfaces for software design
- GUI components, dialog boxes, events, and listeners