Chapter 6: 
Object-Oriented Design

Java Software Solutions
Foundations of Program Design
Sixth Edition

by
Lewis & Loftus
This Week

- Finish Midterm Review
- Static classes and methods
- Cloning
- Interfaces
This Week

• Methods
  – Method Overloading
  – Method Decomposition

• BorderLayout
• Describe IRP
• Describe Project 2
Static Variables and Methods
Static Class Members

• Recall that a static method is one that can be invoked through its class name

• For example, the methods of the `Math` class are static:

  ```java
  result = Math.sqrt(25)
  ```

• Variables can be static as well

• Determining if a method or variable should be static is an important design decision
The static Modifier

- We declare static methods and variables using the static modifier.
- It associates the method or variable with the class rather than with an object of that class.
- Static methods are sometimes called class methods and static variables are sometimes called class variables.
- Let's carefully consider the implications of each.
Static Variables

• Normally, each object has its own data space, but 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 is first referenced

• All objects instantiated from the class share its static variables

• Changing the value of a static variable in one object changes it for all others
Static Methods

class Helper
{
    public static int cube (int num)
    {
        return num * num * num;
    }
}

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

    value = Helper.cube(5);
Static Class Members

• The order of the modifiers can be interchanged, but by convention visibility modifiers come first.

• Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object.

• Static methods cannot reference instance variables because instance variables don't exist until an object exists.

• However, a static method can reference static variables or local variables.
Static Class Members

• Static methods and static variables often work together

• The following example keeps track of how many `Slogan` objects have been created using a static variable, and makes that information available using a static method

• See `SloganCounter.java`
• See `Slogan.java`
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`, which is invoked as follows:

```java
obj1.tryMe();
obj2.tryMe();
```

In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`. 
The this reference

- The this reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names.

- The constructor of the Account class (from Chapter 4) could have been written as follows:

```java
public Account (String name, long acctNumber, double balance)
{
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
}
```
Cloning
Cloning

• Copying a primitive type is simple
  – int x = 10;
  – int y = x; // Creates a copy

• Copying a complex type (object) isn’t so simple.

• See samplecode/cloning
Naïve Copy

- Given:

```java
public class Student {
    private int GPA;
    private Address address;
    private int numCredits;

    public Student(int GPA, int numCredits, String street,
                    String city, String state) {
        this.numCredits = numCredits;
        this.GPA = GPA;
        this.address = new Address(street, city);
    }
}

Student s1 = new Student( ... );
Student s2 = s1; // What does this do?
```
Naïve Copy

\[
s_2 = s_1;
\]
s2 = s1.clone();

The default operation for clone is a shallow copy. Copies each class attribute.
Best: custom clone();

s2 = s1.clone();

You must create your own deep copy clone method!
Interfaces
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 a set of methods that a class will implement
Understanding Interfaces

Interface is:
- Steering wheel
- Gas pedal

Hybrid

Traditional

BUT....

How is it implemented?
WHO CARES! 😊
Interfaces

interface is a reserved word

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

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

A semicolon immediately follows each method header

Doable.java
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
Interfaces

```java
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.
Interfaces

What are some interfaces we’ve already used?
Interfaces

• A class that implements an interface can implement other methods as well

• See Complexity.java
• See Question.java
• See MiniQuiz.java

• 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
}
```
Interfaces

• The Java standard class library contains many helpful interfaces

• The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects

• We discussed the `compareTo` method of the `String` class in Chapter 5

• The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order
The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

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

The value returned from `compareTo` should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`.

When a programmer designs a class that implements the `Comparable` interface, it should follow this intent.
The Comparable Interface

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

• For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number

• The implementation of the method can be as straightforward or as complex as needed for the situation
The Iterator Interface

- As we discussed in Chapter 5, an iterator is an object that provides a means of processing a collection of objects one at a time.

- An iterator is created formally by implementing the `Iterator` interface, which contains three methods:
  - 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 Iterator Interface

• By implementing the `Iterator` interface, a class formally establishes that objects of that type are iterators

• The programmer must decide how best to implement the iterator functions

• Once established, the for-each version of the `for` loop can be used to process the items in the iterator
Interfaces

• You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`).

• However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways.

• Interfaces are a key aspect of object-oriented design in Java.

• We discuss this idea further in Chapter 9.
Method Design
Method Design

• An *algorithm* is a step-by-step process for solving a problem

• Examples: a recipe, travel directions

• Every method implements an algorithm that determines how the method accomplishes its goals

• An algorithm may be expressed in *pseudocode*, a mixture of code statements and English that communicate the steps to take
Method Decomposition

• Small - so that it can be understood as a single entity

• A potentially large method should be decomposed into several smaller methods as needed for clarity

• A public service method of an object may call one or more private support methods to help it accomplish its goal

• Support methods might call other support methods if appropriate
Method Decomposition

• Let's look at an example that requires method decomposition – translating English into Pig Latin

• Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"

• Words that begin with vowels have the "yay" sound added on the end

  book ➔ ookbay  table ➔ abletay
  item ➔ itemyay  chair ➔ airchay
Method Decomposition

• The primary objective (translating a sentence) is too complicated for one method to accomplish

• Therefore we look for natural ways to decompose the solution into pieces

• Translating a sentence can be decomposed into the process of translating each word

• The process of translating a word can be separated into translating words that:
  – begin with vowels
  – begin with consonant blends (sh, cr, th, etc.)
  – begin with single consonants
Psuedocode

• Lets write pseudocode for the translator
Method Decomposition

• See PigLatin.java
• See PigLatinTranslator.java

• In a UML class diagram, the visibility of a variable or method can be shown using special characters
  • Public members are preceded by a plus sign
  • Private members are preceded by a minus sign
Class Diagram for Pig Latin

- PigLatin
  + main (args : String[]) : void

  PigLatinTranslator
  + translate (sentence : String) : String
  - translateWord (word : String) : String
  - beginsWithVowel (word : String) : boolean
  - beginsWithBlend (word : String) : boolean
Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions.

- If a method is overloaded, the method name is not sufficient to determine which method is being called.

- The *signature* of each overloaded method must be unique.

- The signature includes the number, type, and order of the parameters.
Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```java
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}

Invocation

result = tryMe(25, 4.32)
```
Method Overloading

- The `println` method is overloaded:

  ```java
  println (String s)
  println (int i)
  println (double d)
  ```

  and so on...

- The following lines invoke different versions of the `println` method:

  ```java
  System.out.println ("The total is:");
  System.out.println (total);
  ```
Overloading Methods

• The return type of the method is not part of the signature

• That is, overloaded methods cannot differ only by their return type

• Constructors can be overloaded

• Overloaded constructors provide multiple ways to initialize a new object
Testing Terminology

• The goal of testing is to gain confidence that our program works as planned and find any deviations from our expectations.

• Black box testing
  • In \textit{black-box testing}, test cases are developed without considering the internal logic
  • They are based on the input and expected output

• White box testing
  • \textit{White-box testing} focuses on the internal structure of the code
  • The goal is to ensure that every path through the code is tested
Review Questions

• Explain the Comparable interface
• What is method overloading?
• What is an interface? Can one class “use” two of them?

• What is the IRP?
Border Layout
Layout Managers

- A *layout manager* is an object that determines the way that components are arranged in a container.

- There are several predefined layout managers defined in the Java standard class library:
  - Defined in the AWT:
    - Flow Layout
    - Border Layout
    - Card Layout
    - Grid Layout
    - GridBag Layout
  - Defined in Swing:
    - Box Layout
    - Overlay Layout
Layout Managers

- Every container has a default layout manager, but we can explicitly set the layout manager as well.

- Each layout manager has its own particular rules governing how the components will be arranged.

- Some layout managers pay attention to a component's preferred size or alignment, while others do not.

- A layout manager attempts to adjust the layout as components are added and as containers are resized.
Border Layout

- A *border layout* defines five areas into which components can be added

![Diagram of border layout with five areas: North, South, West, Center, East]
Border Layout

• Each area displays one component (which could be a container such as a JPanel)

• Each of the four outer areas enlarges as needed to accommodate the component added to it

• If nothing is added to the outer areas, they take up no space and other areas expand to fill the void

• The center area expands to fill space as needed

• See BorderPanel.java