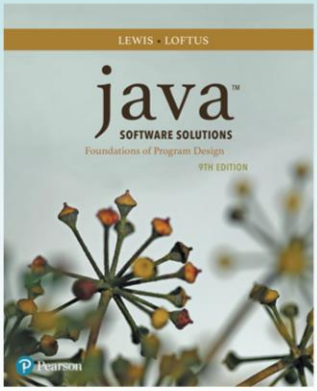
Chapter 7
Object-Oriented Design



Java Software Solutions
Foundations of Program Design
9th Edition

John Lewis
William Loftus

Object-Oriented Design

- Now we can extend our discussion of the design of classes and objects
- Chapter 7 focuses on:
  - software development activities
  - determining the classes and objects that are needed for a program
  - the relationships that can exist among classes
  - the static modifier
  - writing interfaces
  - the design of enumerated type classes
  - method design and method overloading

Outline

- Software Development Activities
- Identifying Classes and Objects
- Static Variables and Methods
- Class Relationships
- Interfaces
- Enumerated Types Revisited
- **Method Design and Overloading**
- Testing

Method Design

- As we've discussed, high-level design issues include:
  - identifying primary classes and objects
  - assigning primary responsibilities
- After establishing high-level design issues, its important to address low-level issues such as the design of key methods
- For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design

-Implementing methods begins with thinking about the steps involved to accomplish the desired goal

-We often refer to these steps as an **algorithm**

-An algorithm defines the logic and statements necessary to carry out the goal of the method

Method Decomposition

- A method should be relatively 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

-By decomposing a large method into a set of smaller ones, the smaller ones might be reused by other methods

-We are always looking for ways to **reuse** code (classes, methods) in effective object-oriented design

Method Decomposition

- 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
- See `PigLatin.java`
- See `PigLatinTranslator.java`

```
//********************************************************************
//  PigLatin.java       Author: Lewis/Loftus
//
//  Demonstrates the concept of method decomposition.
//********************************************************************

import java.util.Scanner;

public class PigLatin
{
   //-----------------------------------------------------------------
   //  Reads sentences and translates them into Pig Latin.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      String sentence, result, another;

      Scanner scan = new Scanner (System.in);
```

continue

continue

```
      do
      {
         System.out.println ();
         System.out.println ("Enter a sentence (no punctuation):");
         sentence = scan.nextLine();

         System.out.println ();
         result = PigLatinTranslator.translate (sentence);
         System.out.println ("That sentence in Pig Latin is:");
         System.out.println (result);

         System.out.println ();
         System.out.print ("Translate another sentence (y/n)? ");
         another = scan.nextLine();
      }
      while (another.equalsIgnoreCase("y"));
   }
}
```

continue

```
      do
      {
         Syst
         Syst                                    ation):");
         sent

         Syst
         resu                                    ;
         Syst                                    is:");
         Syst

         Syst
         Syst                                    /n)? ");
         anot
      }
      while (
   }
}
```

Sample Run

```
Enter a sentence (no punctuation):
Do you speak Pig Latin

That sentence in Pig Latin is:
oday ouyay eakspay igpay atinlay

Translate another sentence (y/n)? y

Enter a sentence (no punctuation):
Play it again Sam

That sentence in Pig Latin is:
ayplay ityay againyay amsay

Translate another sentence (y/n)? n
```

```
//********************************************************************
//  PigLatinTranslator.java       Author: Lewis/Loftus
//
//  Represents a translator from English to Pig Latin. Demonstrates
//  method decomposition.
//********************************************************************

import java.util.Scanner;

public class PigLatinTranslator
{
   //-----------------------------------------------------------------
   //  Translates a sentence of words into Pig Latin.
   //-----------------------------------------------------------------
   public static String translate (String sentence)
   {
      String result = "";

      sentence = sentence.toLowerCase();

      Scanner scan = new Scanner (sentence);

      while (scan.hasNext())
      {
         result += translateWord (scan.next());
         result += " ";
      }
```

continue

continue

```
      return result;
   }

   //-----------------------------------------------------------------
   //  Translates one word into Pig Latin. If the word begins with a
   //  vowel, the suffix "yay" is appended to the word.  Otherwise,
   //  the first letter or two are moved to the end of the word,
   //  and "ay" is appended.
   //-----------------------------------------------------------------
   private static String translateWord (String word)
   {
      String result = "";

      if (beginsWithVowel(word))
         result = word + "yay";
      else
         if (beginsWithBlend(word))
            result = word.substring(2) + word.substring(0,2) + "ay";
         else
            result = word.substring(1) + word.charAt(0) + "ay";

      return result;
   }
```

continue

-Although not necessary, note how some of the methods have private visibility

-This means they can **only** be called by methods within the class

-They cannot be called by objects from outside the class (e.g. in a main method of a driver class)

continue

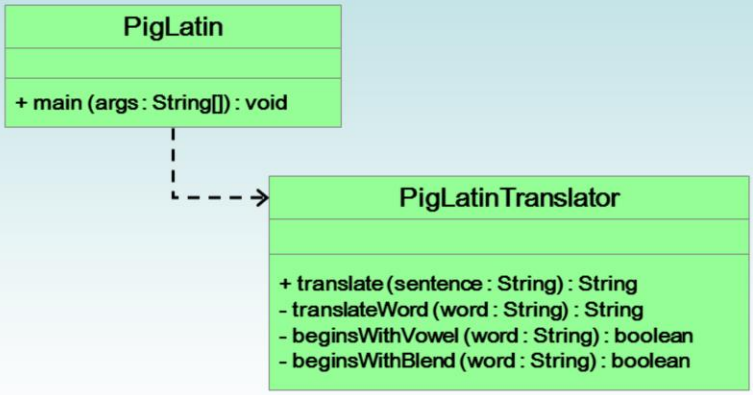
```
//-----------------------------------------------------------------
//  Determines if the specified word begins with a vowel.
//-----------------------------------------------------------------
private static boolean beginsWithVowel (String word)
{
   String vowels = "aeiou";

   char letter = word.charAt(0);

   return (vowels.indexOf(letter) != -1);
}
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Determines if the specified word begins with a particular
   //  two-character consonant blend.
   //-----------------------------------------------------------------
   private static boolean beginsWithBlend (String word)
   {
      return ( word.startsWith ("bl") || word.startsWith ("sc") ||
               word.startsWith ("br") || word.startsWith ("sh") ||
               word.startsWith ("ch") || word.startsWith ("sk") ||
               word.startsWith ("cl") || word.startsWith ("sl") ||
               word.startsWith ("cr") || word.startsWith ("sn") ||
               word.startsWith ("dr") || word.startsWith ("sm") ||
               word.startsWith ("dw") || word.startsWith ("sp") ||
               word.startsWith ("fl") || word.startsWith ("sq") ||
               word.startsWith ("fr") || word.startsWith ("st") ||
               word.startsWith ("gl") || word.startsWith ("sw") ||
               word.startsWith ("gr") || word.startsWith ("th") ||
               word.startsWith ("kl") || word.startsWith ("tr") ||
               word.startsWith ("ph") || word.startsWith ("tw") ||
               word.startsWith ("pl") || word.startsWith ("wh") ||
               word.startsWith ("pr") || word.startsWith ("wr") );
   }
}
```

Class Diagram for Pig Latin



Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the *actual parameter* (the value passed in) is stored into the *formal parameter* (in the method header)
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

-**Pass by value:** actual values are copied to the parameter variable in the method

-Passing parameters have different affects depending on the data type of the parameter being passed:

Primitive variables

-If the method modifies the variable, the original parameter value (from the calling method) is **not** changed!

Object reference variables

-If the method changes the variable, the original parameter (from the calling method) is changed!

-Why the difference? Because the value copied with object variables is the **address** of the object

-As a result, both the variable passed and copied both point to the same object in memory!

Passing Objects to Methods

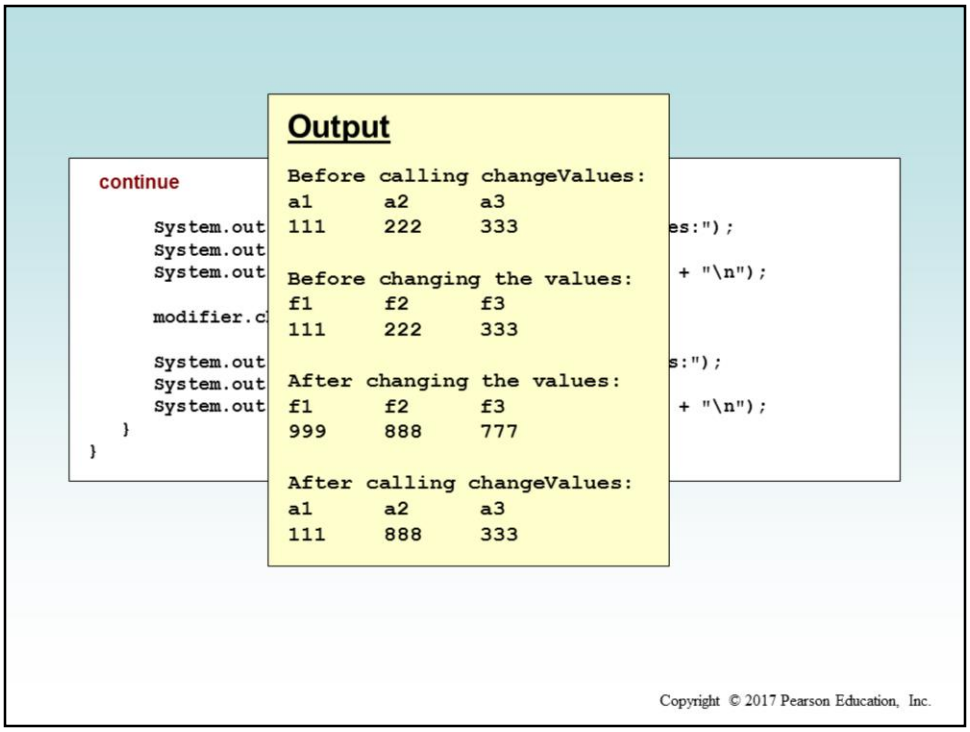
- What a method does with a parameter may or may not have a permanent effect (outside the method)
- Note the difference between changing the internal state of an object versus changing which object a reference points to
- See `ParameterTester.java`
- See `ParameterModifier.java`
- See `Num.java`

```
//********************************************************************
//  ParameterTester.java       Author: Lewis/Loftus
//
//  Demonstrates the effects of passing various types of parameters.
//********************************************************************

public class ParameterTester
{
   //-----------------------------------------------------------------
   //  Sets up three variables (one primitive and two objects) to
   //  serve as actual parameters to the changeValues method. Prints
   //  their values before and after calling the method.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      ParameterModifier modifier = new ParameterModifier();

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

continue

continue

```
        System.out.println ("Before calling changeValues:");
        System.out.println ("a1\ta2\ta3");
        System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

        modifier.changeValues (a1, a2, a3);

        System.out.println ("After calling changeValues:");
        System.out.println ("a1\ta2\ta3");
        System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
    }
}
```



-Since a1 is a primitive type (int), changing the copy in the method **does not** change the original value

-Since a2 is an object variable, changing its state (values of instance variables) in the method **does** change the original object

-With a3, we are reassigning a different object to the copied reference variable in the method, but the original object reference variable **does not** change

-With a3, the copy in the method points to a different object, the original variable still points to the original object

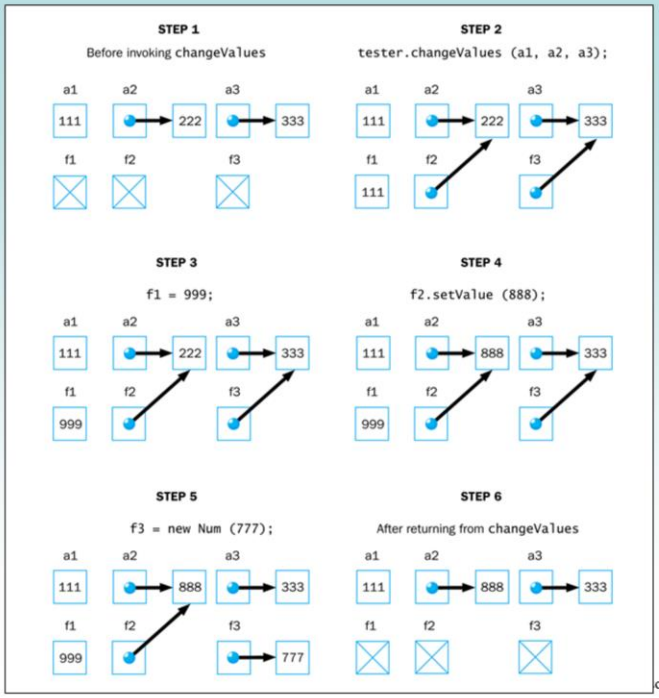
```
//********************************************************************
//  ParameterModifier.java       Author: Lewis/Loftus
//
//  Demonstrates the effects of changing parameter values.
//********************************************************************

public class ParameterModifier
{
   //-----------------------------------------------------------------
   //  Modifies the parameters, printing their values before and
   //  after making the changes.
   //-----------------------------------------------------------------
   public void changeValues (int f1, Num f2, Num f3)
   {
      System.out.println ("Before changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

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

      System.out.println ("After changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
   }
}
```

```
//********************************************************************
//  Num.java       Author: Lewis/Loftus
//
//  Represents a single integer as an object.
//********************************************************************

public class Num
{
   private int value;

   //-----------------------------------------------------------------
   //  Sets up the new Num object, storing an initial value.
   //-----------------------------------------------------------------
   public Num (int update)
   {
      value = update;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Sets the stored value to the newly specified value.
   //-----------------------------------------------------------------
   public void setValue (int update)
   {
      value = update;
   }

   //-----------------------------------------------------------------
   //  Returns the stored integer value as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      return value + "";
   }
}
```



on Education, Inc.

Method Overloading

- Let's look at one more important method design issue: method overloading
- *Method overloading* is the process of giving a single method name multiple definitions in a class
- 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 is typically used to perform the same behaviors (services) with different data types

-The method names are the same but the variable types and number of parameters (signature) differ

-Constructors are examples where we see overloading

-Multiple constructors can be written with different signatures

-Overloaded methods **cannot** differ only by the variable type return

-For example, the following two methods would cause an error:

int sum(int)

double sum(int)

Method Overloading

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

```
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:

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

and so on...

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

```
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

Outline

- Software Development Activities
- Identifying Classes and Objects
- Static Variables and Methods
- Class Relationships
- Interfaces
- Enumerated Types Revisited
- Method Design
- Testing

Testing

- Testing can mean many different things
- It certainly includes running a completed program with various inputs
- It also includes any evaluation performed by human or computer to assess quality
- Some evaluations should occur before coding even begins
- The earlier we find an problem, the easier and cheaper it is to fix

Testing

- The goal of testing is to find errors
- As we find and fix errors, we raise our confidence that a program will perform as intended
- We can never really be sure that all errors have been eliminated
- So when do we stop testing?
  - Conceptual answer: Never
  - Cynical answer: When we run out of time
  - Better answer: When we are willing to risk that an undiscovered error still exists

Reviews

- A *review* is a meeting in which several people examine a design document or section of code
- It is a common and effective form of human-based testing
- Presenting a design or code to others:
  - makes us think more carefully about it
  - provides an outside perspective
- Reviews are sometimes called *inspections* or *walkthroughs*

Test Cases

- A *test case* is a set of input and user actions, coupled with the expected results
- Often test cases are organized formally into *test suites* which are stored and reused as needed
- For medium and large systems, testing must be a carefully managed process
- Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

- *Defect testing* is the execution of test cases to uncover errors
- The act of fixing an error may introduce new errors
- After fixing a set of errors we should perform *regression testing* – running previous test suites to ensure new errors haven't been introduced
- It is not possible to create test cases for all possible input and user actions
- Therefore we should design tests to maximize their ability to find problems

Black-Box Testing

- In *black-box testing*, test cases are developed without considering the internal logic
- They are based on the input and expected output
- Input can be organized into *equivalence categories*
- Two input values in the same equivalence category would produce similar results
- Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories

White-Box Testing

- *White-box testing* focuses on the internal structure of the code
- The goal is to ensure that every path through the code is tested
- Paths through the code are governed by any conditional or looping statements in a program
- A good testing effort will include both black-box and white-box tests