Method Design & Method Overloading

November 8, 2006

Quick Review of Last Lecture

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

The this reference

```java
public Account (String name, long acctNumber, double balance) {
    name = owner;
    acctNumber = account;
    balance = initial;
}
```

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

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

interface is a reserved word

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

A semicolon immediately follows each method header

```
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

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

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

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

A class that implements an interface can implement other methods as well
• See Complexity.java (page 310)
• See Question.java (page 311)
• See MiniQuiz.java (page 313)
• 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

- 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

Where can you find the standard Java interfaces

- C:\Program Files\Java\jdk1.5.0\src.zip

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 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.

Interface Example:

- `Sortable.java`
- `SortableIntArray.java`
- `SortableStrigArray.java`
- `SortingTest.java`

Chapter 6
Section 6.6

Enumerated Types

(read Section 6.6 on your own)

Chapter 6
Section 6.7

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, it’s 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.
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

- 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 → aircay

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

Method Decomposition

- See PigLatin.java (page 320)
- See PigLatinTranslator.java (page 323)
- 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
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

```java
float tryMe(int x)
{
    return x + .375;
}
float tryMe(int x, float y)
{
    return x*y;
}
result = tryMe(25, 4.32)
```

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
```

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

```java
System.out.println ("The total is:");
System.out.println (total);
```
Testing (read Section 6.9 on your own)

- 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
  - Snide 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.