Interfaces

November 6, 2006

Quick Review of Last Lecture

Objects – instances of a class with a static variable ‘size’

Objects – instances of classes

• Note that the variables can have different values in the two objects

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

• 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:

result = Math.sqrt(25);

• Variables can be static as well
• Determining if a method or variable should be static is an important design decision
Static Methods

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

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

```java
value = Helper.cube(5);
```

Method Control Flow

- If the called method is in the same class, only the method name is needed.

Accessing Variables

- If the called method is in the same class, only the method name is needed.

```java
int myVariable;
myVariable=5;
```

- Static methods cannot use non static class variables.

```java
int myVariable;
myVariable=5;
```

- Static methods can use static class variables.

```java
static int myVariable;
myVariable=5;
```

- Static methods can only call other static methods within the same class.

```java
main static cube static helpMe
helper.cube();
helpMe();
```
Chapter 6
Section 6.4

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A uses B
  - Aggregation: A has-a B
  - Inheritance: A is-a B

Dependency

- A dependency exists when one class relies on another in some way, usually by invoking the methods of the other
- We’ve seen dependencies in many previous examples
- We don’t want numerous or complex dependencies among classes
- Nor do we want complex classes that don’t depend on others
- A good design strikes the right balance

Dependency Example: Client-Server

- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the concat method of the String class takes as a parameter another String object
  ```java
  str3 = str1.concat(str2);
  ```
- This drives home the idea that the service is being requested from a particular object

Concatenation Example
Dependency

- The following example defines a class called Rational to represent a rational number
- A rational number is a value that can be represented as the ratio of two integers
- Some methods of the Rational class accept another Rational object as a parameter
- See RationalTester.java (page 297)
- See Rational.java (page 299)

Representing Rational Numbers

- public class RationalNumber
  {
    private int numerator, denominator;
    // ...
  }

Adding Two rational numbers

public RationalNumber add (RationalNumber op2)
{
  int commonDenominator = denominator * op2.getDenominator();
  int numerator1 = numerator * op2.getDenominator();
  int numerator2 = op2.getNumerator() * denominator;
  int sum = numerator1 + numerator2;
  return new RationalNumber (sum, commonDenominator);
}

Aggregation

- An aggregate is an object that is made up of other objects
- Therefore aggregation is a has-a relationship
  - A car has a chassis
  - A student has an address

Aggregation Example: Components of a Student
Aggregation

- In the following example, a student object is composed, in part, of address objects.
- A student has an address (in fact each student has two addresses).
- See StudentBody.java (page 304)
- See Student.java (page 306)
- See Address.java (page 307)
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end.

A More Complicated Student Example

Other Stuff from Section 6.4
How would you write the code for the more complicated student example?

Inheritance is discussed in Chapter 8

Abstract

Person

Man

Woman

Abstract

Home

Condo

Mansion

5 bedroom
	house

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:

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

The this reference

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

public Account (String name, long acctNumber, double balance)
{
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
}
Chapter 6
Section 6.5 – 6.6

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);

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

public class CanDo implements Doable

public void doThis()

{ // whatever
}

public void doThat()

{ // whatever
}

// etc.

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 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
- SortableStringArray.java
- SortingTest.java

Enumerated Types

- In Chapter 3 we introduced enumerated types, which define a new data type and list all possible values of that type.
  ```java
  enum Season {winter, spring, summer, fall}
  ```
- Once established, the new type can be used to declare variables:
  ```java
  Season time;
  ```
- The only values this variable can be assigned are the ones established in the `enum` definition.

Enumerated Types

- An enumerated type definition is a special kind of class.
- The values of the enumerated type are objects of that type.
- For example, `fall` is an object of type `Season`.
- That's why the following assignment is valid:
  ```java
  time = Season.fall;
  ```
Enumerated Types

• An enumerated type definition can be more interesting than a simple list of values
• Because they are like classes, we can add additional instance data and methods
• We can define an enum constructor as well
• Each value listed for the enumerated type calls the constructor
  • See Season.java (page 318)
  • See SeasonTester.java (page 319)

Enumerated Types

• Every enumerated type contains a static method called values that returns a list of all possible values for that type
• The list returned from values is an iterator, so a for loop can be used to process them easily
• An enumerated type cannot be instantiated outside of its own definition
• A carefully designed enumerated type provides a versatile and type-safe mechanism for managing data

THE END