Module 7

Polymorphism, Abstract Classes, and Interfaces

Adapted from Absolute Java, Rose Williams, *Binghamton University*

Introduction to Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
- Polymorphism is the ability to associate many meanings to one method name
  - It does this through a special mechanism known as ***late binding*** or ***dynamic binding***

Late Binding

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled, that is called ***early binding***
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called ***late binding*** or ***dynamic binding***

Late Binding

- Java uses late binding for all methods (except `private`, `final,` and `static` methods)
- Because of late binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined
- For an example, the relationship between a base class called `Sale` and its derived class `DiscountSale` will be examined

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

```
Sale saleVariable; //Base class
DiscountSale discountVariable = new
   DiscountSale("paint", 15,10); //Derived class
saleVariable = discountVariable; //Upcasting
System.out.println(saleVariable.toString());
```

- Because of late binding, **`toString`** above uses the definition given in the **`DiscountSale`** class

Upcasting and Downcasting

- *Downcasting* is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

```
discountVariable =                    //will produce
     (DiscountSale)saleVariable;//run-time error
discountVariable = saleVariable //will produce
                                //compiler error
```

  - There are times, however, when downcasting is necessary, e.g., inside the **`equals`** method for a class:

```
Sale otherSale = (Sale)otherObject;//downcasting
```

No Late Binding for Static Methods

- When the decision of which definition of a method to use is made at compile time, that is called *static binding*
  - This decision is made based on the *type of the variable naming the object*
- Java uses static, not late, binding with private, `final`, and static methods
  - In the case of `private` and `final` methods, late binding would serve no purpose
  - However, in the case of a static method invoked using a calling object, it does make a difference

No Late Binding for Static Methods

- The **`Sale`** class **`announcement()`** method:

```
public static void announcement( )
{
  System.out.println("Sale class");
}
```

- The **`DiscountSale`** class **`announcement()`** method:

```
public static void announcement( )
{
  System.out.println("DiscountSale class");
}
```

No Late Binding for Static Methods

- In the previous example, the **`simple`** (**`Sale`** class) and **`discount`** (**`DiscountClass`**) objects were created
- Given the following assignment:

```
simple = discount;
```

  - Now the two variables point to the same object
  - In particular, a **`Sale`** class variable names a **`DiscountClass`** object

No Late Binding for Static Methods

- Given the invocation:

```
simple.announcement();
```

  - The output is:

```
Sale class
```

- Note that here, **`announcement`** is a static method invoked by a calling object (instead of its class name)
  - Therefore the type of **`simple`** is determined by its variable name, not the object that it references

No Late Binding for Static Methods

- There are other cases where a static method has a calling object in a more inconspicuous way
- For example, a static method can be invoked within the definition of a nonstatic method, but without any explicit class name or calling object
- In this case, the calling object is the implicit `this`

The `final` Modifier

- A *method* marked **`final`** indicates that it cannot be overridden with a new definition in a derived class
  - If **`final`**, the compiler can use early binding with the method

```
public final void someMethod() { . . . }
```

- A *class* marked **`final`** indicates that it cannot be used as a base class from which to derive any other classes

Example: Late Binding with `toString`

- If an appropriate `toString` method is defined for a class, then an object of that class can be output using `System.out.println`

```
Sale aSale = new Sale("tire gauge",
  9.95);
System.out.println(aSale);
```

  - Output produced:

```
tire gauge Price and total cost = $9.95
```

- This works because of late binding

Example: Late Binding with `toString`

- One definition of the method `println` takes a single argument of type `Object`:

```
public void println(Object theObject)
{
  System.out.println(theObject.toString());
}
```

  - In turn, It invokes the version of `println` that takes a `String` argument
- Note that the `println` method was defined before the `Sale` class existed
- Yet, because of late binding, the `toString` method from the `Sale` class is used, not the `toString` from the `Object` class

An Object knows the Definitions of its Methods

- The type of a class variable determines which method names can be used with the variable
  - However, the object named by the variable determines which definition with the same method name is used
- A special case of this rule is as follows:
  - The type of a class parameter determines which method names can be used with the parameter
  - The argument determines which definition of the method name is used

Downcasting

- It is the responsibility of the programmer to use downcasting only in situations where it makes sense
  - The compiler does not check to see if downcasting is a reasonable thing to do
- Using downcasting in a situation that does not make sense usually results in a run-time error

Checking to See if Downcasting is Legitimate

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type
  - This is exactly what the **`instanceof`** operator tests for:

    `object` **`instanceof`** ***`ClassName`***
  - It will return true if ***`object`*** is of type ***`ClassName`***
  - In particular, it will return true if ***`object`*** is an instance of any descendent class of ***`ClassName`***

Introduction to Abstract Classes

- While discussing Inheritance, the **`Employee`** base class and two of its derived classes, **`HourlyEmployee`** and **`SalariedEmployee`** were defined
- The following method is added to the **`Employee`** class
  - It compares employees to to see if they have the same pay:

```
public boolean samePay(Employee other)
{
  return(this.getPay() == other.getPay());
}
```

Introduction to Abstract Classes

- There are several problems with this method:
  - The `getPay` method is invoked in the `samePay` method
  - There are `getPay` methods in each of the derived classes
  - There is no `getPay` method in the `Employee` class, nor is there any way to define it reasonably without knowing whether the employee is hourly or salaried

Introduction to Abstract Classes

- The ideal situation would be if there were a way to
  - Postpone the definition of a `getPay` method until the type of the employee were known (i.e., in the derived classes)
  - Leave some kind of note in the `Employee` class to indicate that it was accounted for
- Surprisingly, Java allows this using abstract classes and methods

Introduction to Abstract Classes

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
- The class that contains an abstract method is called an *abstract class*

Abstract Method

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier **`abstract`**
- It cannot be private
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();
public abstract void doIt(int count);
```

Abstract Class

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **`abstract`** included in its class heading:

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add `abstract` to its modifier

- A class that has no abstract methods is called a *concrete class*

You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of **`super`**

An Abstract Class Is a Type

- Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type
  - This makes it possible to plug in an object of any of its descendent classes
- It is also fine to use a variable of an abstract class type, as long is it names objects of its concrete descendent classes only

Interfaces

- An *interface* is something like an extreme case of an abstract class
  - However, *an interface is not a class*
  - *It is a type that can be satisfied by any class that implements the interface*
- The syntax for defining an interface is similar to that of defining a class
  - Except the word **`interface`** is used in place of **`class`**
- An interface specifies a set of methods that any class that implements the interface must have
  - It contains method headings and constant definitions only
  - It contains no instance variables nor any complete method definitions

Interfaces

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This *multiple inheritance* is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

Interfaces

- An interface and all of its method headings should be declared public
  - They cannot be given private, protected, or package access
- When a class implements an interface, it must make all the methods in the interface public
- Because an interface is a type, a method may be written with a parameter of an interface type
  - That parameter will accept as an argument any class that implements the interface

The **`Ordered`** Interface

```
1  public interface Ordered
2  {
3      public boolean precedes(Object other);

4      /**
5       For objects of the class o1 and o2,
6       o1.follows(o2) == o2.preceded(o1).
7      */
8      public boolean follows(Object other);
9  }
```

Do not forget the semicolons at the end of the method headings.

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied. It is only advisory to the programmer implementing the interface.

Interfaces

- To *implement an interface*, a concrete class must do two things:
  1. It must include the phrase

     `implements` ***`Interface_Name`***

     at the start of the class definition
     - If more than one interface is implemented, each is listed, separated by commas
  1. The class must implement *all* the method headings listed in the definition(s) of the interface(s)
- Note the use of **`Object`** as the parameter type in the following examples

Implementation of an Interface

```
1  public class OrderedHourlyEmployee
2           extends HourlyEmployee implements Ordered
3  {
4      public boolean precedes(Object other)
5      {
6          if (other == null)
7              return false;
8          else if (!(other instanceof HourlyEmployee))
9              return false;
10         else
11         {
12             OrderedHourlyEmployee otherOrderedHourlyEmployee =
13                                 (OrderedHourlyEmployee)other;
14              return (getPay() < otherOrderedHourlyEmployee.getPay());
15         }
16     }
```

Although `getClass` works better than `instanceof` for defining `equals`, `instanceof` works better here. However, either will do for the points being made here.

Implementation of an Interface

```
17      public boolean follows(Object other)
18      {
19          if (other == null)
20              return false;
21          else if (!(other instanceof OrderedHourlyEmployee))
22              return false;
23          else
24          {
25              OrderedHourlyEmployee otherOrderedHourlyEmployee =
26                                  (OrderedHourlyEmployee)other;
27              return (otherOrderedHourlyEmployee.precedes(this));
28          }
29      }
30  }
```

Abstract Classes Implementing Interfaces

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

An Abstract Class Implementing an Interface

```
1   public abstract class MyAbstractClass implements Ordered
2   {
3       int number;
4       char grade;
5
6       public boolean precedes(Object other)
7       {
8           if (other == null)
9               return false;
10          else if (!(other instanceof HourlyEmployee))
11              return false;
12          else
13          {
14              MyAbstractClass otherOfMyAbstractClass =
15                                              (MyAbstractClass)other;
16              return (this.number < otherOfMyAbstractClass.number);
17          }
18      }

19      public abstract boolean follows(Object other);

20  }
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase

    ```
    extends BaseInterfaceName
    ```
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

Extending an Interface

```
1  public interface ShowablyOrdered extends Ordered
2  {
3      /**
4        Outputs an object of the class that precedes the calling object.
5      */
6      public void showOneWhoPrecedes();
7  }
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the `ShowablyOrdered` interface must have a definition for the method `showOneWhoPrecedes` and also have definitions for the methods `precedes` and `follows` given in the `Ordered` interface.

Defined Constants in Interfaces

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be public, static, and final
  - Because this is understood, Java allows these modifiers to be omitted
- Any class that implements the interface has access to these defined constants

Inconsistent Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

Inconsistent Interfaces

- When a class implements two interfaces:
  - One type of inconsistency will occur if the interfaces have constants with the same name, but with different values
  - Another type of inconsistency will occur if the interfaces contain methods with the same name but different return types
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is illegal

Inner Classes

- Inner classes are classes defined within other classes
  - The class that includes the inner class is called the outer class
  - There is no particular location where the definition of the inner class (or classes) must be place within the outer class
  - Placing it first or last, however, will guarantee that it is easy to find

Inner Classes

- An inner class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members
  - An inner class is local to the outer class definition
  - The name of an inner class may be reused for something else outside the outer class definition
  - If the inner class is private, then the inner class cannot be accessed by name outside the definition of the outer class

Inner Classes

- There are two main advantages to inner classes
  - They can make the outer class more self-contained since they are defined inside a class
  - Both of their methods have access to each other's private methods and instance variables
- Using an inner class as a helping class is one of the most useful applications of inner classes
  - If used as a helping class, an inner class should be marked private

Inner and Outer Classes Have Access to Each Other's Private Members

- Within the definition of a method of an inner class:
  - It is legal to reference a private instance variable of the outer class
  - It is legal to invoke a private method of the outer class
- Within the definition of a method of the outer class
  - It is legal to reference a private instance variable of the inner class on an object of the inner class
  - It is legal to invoke a (nonstatic) method of the inner class as long as an object of the inner class is used as a calling object
- Within the definition of the inner or outer classes, the modifiers `public` and `private` are equivalent

Class with an Inner Class

```
1  public class BankAccount
2  {
3      private class Money
4      {
5          private long dollars;
6          private int cents;
7          public Money(String stringAmount)
8          {
9              abortOnNull(stringAmount);
10             int length = stringAmount.length();
11             dollars = Long.parseLong(
12                         stringAmount.substring(0, length - 3));
13             cents = Integer.parseInt(
14                         stringAmount.substring(length - 2, length));
15         }
16         public String getAmount()
17         {
18             if (cents > 9)
19                return (dollars + "." + cents);
20             else
21                return (dollars + ".0" + cents);
22         }
```

The modifier `private` in this line should not be changed to `public`.

However, the modifiers `public` and `private` inside the inner class `Money` can be changed to anything else and it would have no effect on the class `BankAccount`.

Class with an Inner Class

```
23      public void addIn(Money secondAmount)
24      {
25          abortOnNull(secondAmount);
26          int newCents = (cents + secondAmount.cents)%100;
27          long carry = (cents + secondAmount.cents)/100;
28          cents = newCents;
29          dollars = dollars + secondAmount.dollars + carry;
30      }

31      private void abortOnNull(Object o)
32      {
33          if (o == null)
34          {
35              System.out.println("Unexpected null argument.");
36              System.exit(0);
37          }
38      }
39  }
```

The definition of the inner class ends here, but the definition of the outer class continues in Part 2 of this display.

Class with an Inner Class

```
40      private Money balance;

41      public BankAccount()
42      {
43          balance = new Money("0.00");
44      }

45      public String getBalance()
46      {
47          return balance.getAmount();
48      }

49      public void makeDeposit(String depositAmount)
50      {
51          balance.addIn(new Money(depositAmount));
52      }

53      public void closeAccount()
54      {
55          balance.dollars = 0;
56          balance.cents = 0;
57      }
58  }
```

To invoke a nonstatic method of the inner class outside of the inner class, you need to create an object of the inner class.

This invocation of the inner class method `getAmount()` would be allowed even if the method `getAmount()` were marked as `private`.

Notice that the outer class has access to the private instance variables of the inner class.

This class would normally have more methods, but we have only included the methods we need to illustrate the points covered here.

The `.class` File for an Inner Class

- Compiling any class in Java produces a `.class` file named ***`ClassName.class`***
- Compiling a class with one (or more) inner classes causes both (or more) classes to be compiled, and produces two (or more) .class files
  - Such as ***`ClassName.class`*** **and** ***`ClassName$InnerClassName.class`***