Chapter 8
Polymorphism and Abstract Classes

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

Introduction to Polymorphism

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes
- Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class

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

The Sale and DiscountSale Classes

- The Sale class contains two instance variables
  - name: the name of an item (String)
  - price: the price of an item (double)
- It contains three constructors
  - A no-argument constructor that sets name to "No name yet", and price to 0.0
  - A two-parameter constructor that takes in a String (for name) and a double (for price)
  - A copy constructor that takes in a Sale object as a parameter
The Sale and DiscountSale Classes

- The Sale class also has a set of accessors (getName, getPrice), mutators (setName, setPrice), overridden equals and toString methods, and a static announcement method
- The Sale class has a method bill, that determines the bill for a sale, which simply returns the price of the item
- It has two methods, equalDeals and lessThan, each of which compares two sale objects by comparing their bills and returns a boolean value

- The DiscountSale class inherits the instance variables and methods from the Sale class
- In addition, it has its own instance variable, discount (a percent of the price), and its own suitable constructor methods, accessor method (getDiscount), mutator method (setDiscount), overwritten toString method, and static announcement method
- The DiscountSale class has its own bill method which computes the bill as a function of the discount and the price

The Sale and DiscountSale Classes

- The Sale class lessThan method
  - Note the bill() method invocations:

  ```java
  public boolean lessThan (Sale otherSale)
  {
      if (otherSale == null)
      {
          System.out.println("Error: null object");
          System.exit(0);
      }
      return (bill( ) < otherSale.bill( ));
  }
  ```

- The DiscountSale class bill() method:

  ```java
  public double bill( )
  {
      double fraction = discount/100;
      return (1 - fraction) * getPrice( );
  }
  ```

The Sale and DiscountSale Classes

- Given the following in a program:

  ```java
  Sale simple = new sale("floor mat", 10.00);
  DiscountSale discount = new DiscountSale("floor mat", 11.00, 10);
  ...
  if (discount.lessThan(simple))
      System.out.println("$" + discount.bill() + 
                      " < "$ + simple.bill() + 
                      " because late-binding works!");
  ```

  - Output would be:

  ```java
  $9.90 < $10 because late-binding works!
  ```

The Sale and DiscountSale Classes

- In the previous example, the boolean expression in the if statement returns true
- As the output indicates, when the lessThan method in the Sale class is executed, it knows which bill() method to invoke
  - The DiscountSale class bill() method for discount, and the Sale class bill() method for simple
- Note that when the Sale class was created and compiled, the DiscountSale class and its bill () method did not yet exist
  - These results are made possible by late-binding
Pitfall: 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.

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");
}

Pitfall: 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

Pitfall: 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

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
- Can be used for efficiency, but usually more for security.
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`:
  ```java
  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

Late Binding with `toString`

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

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):
  ```java
  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 descendant class):
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:
    ```java
    discountVariable = new DiscountSale(); //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:
    ```java
    Sale otherSale = (Sale)otherObject://downcasting
    ```

Pitfall: 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
Tip: 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 or one of its descendant types.
  – This is exactly what the `instanceof` operator tests for:
    ```java
    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 descendant class of `ClassName`

A First Look at the clone Method

• Every object inherits a method named `clone` from the class `Object`
  – The method `clone` has no parameters
  – It is supposed to return a deep copy of the calling object
• However, the inherited version of the method was not designed to be used “as is”
  – Instead, each class is expected to override it with a more appropriate version

A First Look at the clone Method

• The heading for the `clone` method defined in the `Object` class is as follows:
  ```java
  protected Object clone()
  ```
• The heading for a `clone` method that overrides the `clone` method in the `Object` class can differ somewhat from the heading above
  – A change to a more permissive access, such as from `protected` to `public`, is always allowed when overriding a method definition
  – Changing the return type from `Object` to the type of the class being cloned is allowed because every class is a descendant class of the class `Object`
  – This is an example of a covariant return type.
  ```java
  public Sale clone()
  ```
  ```java
  { return new Sale(this); }
  ```
  and another example:
  ```java
  public DiscountSale clone()
  ```
  ```java
  { return new DiscountSale(this); }
  ```

Pitfall: Sometime the clone Method Return Type is Object

• Prior to version 5.0, Java did not allow covariant return types
  – There were no changes whatsoever allowed in the return type of an overridden method
• Therefore, the `clone` method for all classes had `Object` as its return type
  – Since the return type of the `clone` method of the `Object` class was `Object`, the return type of the overriding `clone` method of any other class was `Object` also

• Prior to Java version 5.0, the `clone` method for the `Sale` class would have looked like this:
  ```java
  public Sale clone()
  ```
  ```java
  { return new Sale(this); }
  ```
  and another example:
  ```java
  public DiscountSale clone()
  ```
  ```java
  { return new DiscountSale(this); }
  ```

• Therefore, the result must always be type cast when using a `clone` method written for an older version of Java
  ```java
  Sale copy = (Sale)original.clone();
  ```
Pitfall: Sometimes the clone Method Return Type is Object

- It is still perfectly legal to use Object as the return type for a clone method, even with classes defined after Java version 5.0
  - When in doubt, it causes no harm to include the type cast
  - For example, the following is legal for the clone method of the Sale class:
    ```java
    Sale copy = original.clone();
    ```
  - However, adding the following type cast produces no problems:
    ```java
    Sale copy = (Sale)original.clone();
    ```

Pitfall: Limitations of Copy Constructors

- Although the copy constructor and clone method for a class appear to do the same thing, there are cases where only a clone will work
- For example, given a method badcopy in the class Sale that copies an array of Sales
  - If this array of Sales contains objects from a derived class of Sale (i.e., DiscountSale), then the copy will be a plain Sale, not a true copy
  ```java
  b[i] = new Sale(a[i]); //plain Sale object
  ```
- The reason this works is because the method clone has the same name in all classes, and polymorphism works with method names
- The copy constructors named Sale and DiscountSale have different names, and polymorphism doesn’t work with methods of different names

Introduction to Abstract Classes

- In Chapter 7, the Employee base class and two of its derived classes, HourlyEmployee and SalariedEmployee were defined
- Try adding the following method to the Employee class. Note the error messages in Eclipse.
  ```java
  public boolean samePay(Employee other) {
      return (this.getPay() == other.getPay());
  }
  ```
- 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
- There is no 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
- 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 (see why?)
- It has no method body, and ends with a semicolon in place of its body
  
  ```java
  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:
    ```java
    public abstract class Employee
    {
      private instanceVariables;
      ...
      public abstract double getPay();
      ...
    }
    ```

- 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

Pitfall: 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. This allows initialization of any instance variables of the abstract class.

Tip: 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. Ex:
  ```java
  Employee bob = new SalariedEmployee();
  ...
  public abstract double getPay();
  ```