Chapter 8
Polymorphism and Abstract Classes

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering
Hanyang 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*

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

Display 8.1 The Base Class Sale

```java
public class Sale
{
    private String name; //A nonempty string
    private double price; //nonnegative

    public Sale()
    {
        name = "No name yet";
        price = 0;
    }

    /**
     Precondition: theName is a nonempty string; thePrice is
    */
    public Sale(String theName, double thePrice)
    {
        setName(theName);
        setPrice(thePrice);
    }

    public Sale(Sale originalObject)
    {
        if (originalObject == null)
        {
            System.out.println("Error: null Sale object.");
            System.exit(0);
        }
        //else
        name = originalObject.name;
        price = originalObject.price;
    }

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

    public double getPrice()
    {
        return price;
    }
```

```java
    */
    public void setPrice(double newPrice)
    {
        if (newPrice >= 0)
            price = newPrice;
        else
        {
            System.out.println("Error: Negative price.");
            System.exit(0);
        }
    }

    public String getName()
    {
        return name;
    }

    /**
     Precondition: newName is a nonempty string.
    */
    public void setName(String newName)
    {
        if (newName != null && newName != "")
            name = newName;
        else
        {
            System.out.println("Error: Improper name value.");
            System.exit(0);
        }
    }

    public String toString()
    {
        return (name + " Price and total cost = $" + price);
    }

    public double bill()
    {
        return price;
    }
```

Display 8.1 The Base Class Sale

```
    public boolean equalDeals(Sale otherSale)
    {
        if (otherSale == null)
            return false;
        else
            return (name.equals(otherSale.name)
                && bill() == otherSale.bill());
    }

    /*
     Returns true if the bill for the calling object is less
     than the bill for otherSale; otherwise returns false.
    */
    public boolean lessThan (Sale otherSale)
    {
        if (otherSale == null)
        {
            System.out.println("Error: null Sale object.");
            System.exit(0);
        }
        //else
        return (bill() < otherSale.bill());
    }

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            Sale otherSale = (Sale)otherObject;
            return (name.equals(otherSale.name)
                && (price == otherSale.price));
        }
    }
}
```

When invoked, these methods will use the definition of the method `bill` *that is appropriate for each of the objects.*

Display 8.2 The Derived Class DiscountSale

```
public class DiscountSale extends Sale
{
    private double discount; //A percent of the price. Cannot be negati

    public DiscountSale()
    {
        super();          The meaning would be unchanged if
        discount = 0;     this line were omitted.
    }

    /**
     Precondition: theName is a nonempty string; thePrice is nonnegativ
     theDiscount is expressed as a percent of the price and is nonnegat
    */
    public DiscountSale(String theName,
                        double thePrice, double theDiscount)
    {
        super(theName, thePrice);
        setDiscount(theDiscount);
    }

    public DiscountSale(DiscountSale originalObject)
    {
        super(originalObject);
        discount = originalObject.discount;
    }

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

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

    public double getDiscount()
    {
        return discount;
    }

    /**
     Precondition: Discount is nonnegative.
    */
    public void setDiscount(double newDiscount)
    {
        if (newDiscount >= 0)
            discount = newDiscount;
        else
        {
            System.out.println("Error: Negative discount.");
            System.exit(0);
        }
    }

    public String toString()
    {
        return (getName() + " Price = $" + getPrice()
                + " Discount = " + discount + "%\n"
                + "   Total cost = $" + bill());
    }

    public boolean equals(Object otherObject)
    <The rest of the definition of equals is Self-Test Exercise 4.>
}
```

Display 8.3 Late Binding Demonstration

```
public class LateBindingDemo
{
    public static void main(String[] args)
    {
        Sale simple = new Sale("floor mat", 10.00);//One item at $10.00
        DiscountSale discount = new DiscountSale("floor mat", 11.00, 10);
                              //One item at $11.00 with a 10% discount
        System.out.println(simple);
        System.out.println(discount);

        if (discount.lessThan(simple))
            System.out.println("Discounted item is cheaper.");
        else
            System.out.println("Discounted item is not cheaper.");

        Sale regularPrice = new Sale("cup holder", 9.90);//One item at $9.90
        DiscountSale specialPrice = new DiscountSale("cup holder", 11.00, 10);
                              //One item at $11.00 with a 10% discount
        System.out.println(regularPrice);
        System.out.println(specialPrice);

        if (specialPrice.equalDeals(regularPrice))
            System.out.println("Deals are equal.");
        else
            System.out.println("Deals are not equal.");
    }
}
```

The method **lessThan** *uses different definitions for* **discount.bill()** *and* **simple.bill()**.

The method **equalDeals** *uses different definitions for* **specialPrice.bill()** *and* **regularPrice.bill()**.

The **equalDeals** *method says that two items are equal provided they have the same name and the same bill (same total cost). It does not matter how the bill (the total cost) is calculated.*

Sample Dialogue

```
floor mat Price and total cost = $10.0
floor mat Price = $11.0 Discount = 10.0%
   Total cost = $9.9
Discounted item is cheaper.
cup holder Price and total cost = $9.9
cup holder Price = $11.0 Discount = 10.0%
   Total cost = $9.9
Deals are equal.
```

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 **`Sale`** and **`DiscountSale`** Classes

- 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`**), overriden **`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:

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

The **`Sale`** and **`DiscountSale`** Classes

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

```
public double bill( )
{
  return price;
}
```

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

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

The `Sale` and `DiscountSale` Classes

- Given the following in a program:

```
   . . .
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:

```
$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

Display 8.4 No Late Binding with Static Methods

```
public class StaticMethodsDemo
{
    public static void main(String[] args)
    {
        Sale.announcement();
        DiscountSale.announcement();
        System.out.println(
             "That showed that you can override a static method definition.");

        Sale s = new Sale();
        DiscountSale discount = new DiscountSale();
        s.announcement();
        discount.announcement();
        System.out.println("No surprises so far, but wait.");

        Sale discount2 = discount;
        System.out.println(
              "discount2 is a DiscountSale object in a Sale variable.");
        System.out.println( "Which definition of announcement() will it use?");
        discount2.announcement();
        System.out.println(
              "It used the Sale version of announcement()!");
    }
}
```

Java uses static binding with static methods so the choice of which definition of a static method to use is determined by the type of the variable, not by the object.

discount and discount2 name the same object, but one is a variable of type Sale and one is a variable of type DiscountSale

Pitfall: No Late Binding for Static Methods

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

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

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

```java
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

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

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

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
```

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
  - 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

- In Chapter 7, 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 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 one 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*

Display 8.7 Employee Class as an Abstract Class

```java
public abstract class Employee
{
    private String name;
    private Date hireDate;

    public abstract double getPay();

    public Employee()
    {
        name = "No name";
        hireDate = new Date("January", 1, 1000); //Just a placeholder.
    }

    public boolean samePay(Employee other)
    {
        if (other == null)
        {
            System.out.println("Error: null Employee object.");
            System.exit(0);
        }
        //else
        return (this.getPay() == other.getPay());
    }

<All other constructor and other method definitions are exactly the same as in Display 7.2.
In particular, they are not abstract methods.>
}
```

The class Date is defined in Display 4.13, but the details are not relevant to the current discussion of abstract methods and classes. There is no need to review the definition of the class Date.

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`