Intermediate Programming

Lecture #8 – Polymorphism and Abstract Classes

Polymorphism

- Polymorphism allows the programmer to write several different methods with the same name.
- Up until now, this has involved methods in the same class that differ by their respective parameter lists.

Polymorphism and Late Binding

- In Java, polymorphism includes the ability for the base and derived classes to have methods of the same name and to have the derived class's methods used by methods belonging to the base class.
- ***Binding*** refers to the process of associating a method definition (where we write its code) with a method invocation (where we call it).
- ***Late binding*** refers when this association is made when the programming is running (known as ***run time***).

`final`

- We have used the reserved word **`final`** so far to indicate variables whose values are not allowed to change, but this is not the only usage:
  - A final method cannot be overriden.
  - A final class cannot be used as a base class for other (derived) classes.

`Final`: Some Examples

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

```
public final class SomeClass {
… …
}
```

Sale.java

```
// Class for a simple sale of one item with no
// tax, no discount or other adjustments.
// Class invariants: The price is always
//                   non-negative:
//                  The name is a nonempty string.

public class Sale {
  private String name;   // A non-empty string
  private double price;  // non-negative

  // Sale() - A defaul construcctor
  public Sale () {
    name = "No name yet";
    price = 0;
  }
```

```
// Sale() - A conversion constructor
// Preconditions: theName is a nonempty string
//                the Price is non-negative
public Sale(String theName, double thePrice) {
  setName(theName);
  setPrice(thePrice);
}

// Sale() - A copy constructor
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.");
}

// getName() - an accessor
public String getName() {
  String newName = new String(name);
  return newName;
}

// getPrice() - an accessor
public double getPrice() {
  return price;
}
```

```
// setName() - a mutator
// Precondition - name 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);
  }
}
```

```
//setPrice() - A mutator
// Precondition: newPrice is non-negative
public void setPrice(double newPrice) {
  if (newPrice >= 0 )
    price = newPrice;
  else {
    System.out.println
                  ("Error: negative price.");
    System.exit(0);
  }
}

// toString() - returns the name and formatted
//              price as a string
public String toString() {
  return (name + " Price and total cost = $ "
              + String.format("%4.2f", price));
}
```

```
// bill() - returns the price of the item
public double bill() {
  return price;
}

// equalsDeals() - Returns true if the names
//      are the same and the bill for the
//      calling object is equal to the bill for
//      otherSale. Otherwise returns false.
//    Also returns false if otherObject is null
public boolean equalDeals(Sale otherSale)  {
  if (otherSale == null)
    return false;
  else
    return (name.equals(otherSale.name)
              && bill() == otherSale.bill());
  // will use the appropriate bill() method for
  // the other object
}
```

```
// lessThan() - Returns true if the bill for
//              the callingObject is less than
//              the bill for otherSale;
//              otherwise returns false
// Precondition: otherSale is not null.
public boolean lessThan(Sale otherSale)  {
  if (otherSale == null) {
    System.out.println
              ("Error: null Sale object.");
    System.exit(0);
  }
  // else
    return (bill() < otherSale.bill());
  // will use the appropriate bill() method for
  // the other object
}
```

```
  // equals() - Returns true if the objects are
  //            equal
  public boolean equals(Object otherObject) {
    // If otherObject is null, they can't be
    // equals
    if (otherObject == null)
      return false;
    // getClass() returns a representation of the
    // object's class we can use it to compare
    // two object to see if they are of the same
    // class. If they aren't, they can't be
    // equal.
```

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

DiscountSale.java

```
// Class for a sale of one item with discount
// expressed as a percent of the price but no
// other adjustments.
// Class invariants: the price is non-negative,
//                   the name is a nonempty
  string
//                   the discount is non-negative
public class DiscountSale extends Sale {
  private double discount; // A percent of the
                           // price
                           // Cannot be negative

  // DiscountSale() - a default constructor
  public DiscountSale() {
    super();
    discount = 0;
  }
```

```
  // DiscountSale() - a conversion constructor
  // Preconditions: theName is a nonempty string,
  //                thePrice is non-negative/
  //                theDiscount is a percent of
  price
  //                and is non-negative
  public DiscountSale(String theName,
                      double thePrice,
                      double theDiscount) {
    super(theName, thePrice);
    setDiscount(theDiscount);
  }

  // DiscountSale() - A copy constructor
  public DiscountSale
           (DiscountSale originalObject) {
    super(originalObject);
    discount = originalObject.discount;
  }
```

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

// bill() - Returns the discounted price of the
item
public double bill()  {
  double fraction = discount / 100;
  return (1 - fraction) * getPrice();
}

// getDiscount() - An accessor
public double getDiscount() {
  return discount;
}
```

```
// setDiscount() - A mutator
// Precondition: Discount is a non-negative
//               percent
public void setDiscount(double newDiscount) {
  if (newDiscount >= 0)
    discount = newDiscount;
  else {
    System.out.println
            ("Error: Negative discount.");
    System.exit(0);
  }
}
```

```
// toString() - returns the name and formatted
//              price and discount as a string
public String toString() {
  return (getName() + " Price $ "
    + String.format("%4.2f", getPrice())
    + " Discount = " + discount + "%\n"
    + "Total cost = $"
    + String.format("%4.2f", bill()));
}
```

```
// equals() - Returns true if the objects are
//            equal
public boolean equals(Object otherObject) {
  // If otherObject is null, they can't be
  // equals
  if (otherObject == null)
    return false;

  // getClass() returns a representation of the
  // object's class.  We can use it to compare
  // two object to see if they are of the same
  // class. If they aren't, they can't be
  // equal.
  else if (getClass() !=
otherObject.getClass())
    return false;
```

```
    else {
      DiscountSale otherSale
                 = (DiscountSale)otherObject;
      return
      (getName().equals(otherSale.getName())
        && (getPrice() == otherSale.getPrice())
        && discount == otherSale.discount);
    }
  }
}
```

LateBinding.java

```
// Demonstrates late binding
public class LateBinding  {

  public static void main(String[] args)  {
    // One item at $10
    Sale simple = new Sale("floor mat", 10.00);
    // One item at $11 with a 10% discount
    DiscountSale discount
      = new DiscountSale("floor mat", 11.00, 10);

    System.out.println(simple);
    System.out.println(discount);
```

```
    // lessThan will use different versions of
    // bill()
    if (discount.lessThan(simple))
      System.out.println
             ("Discounted item is cheaper.");
    else
      System.out.println
             ("Discounted item is not cheaper.");

    // One item at $10
    Sale regularPrice
          = new Sale("cup holder", 9.90);
    // One item at $11 with a 10% discount
    DiscountSale specialPrice
     = new DiscountSale("cup holder", 11.00, 10);

    System.out.println(regularPrice);
    System.out.println(specialPrice);
```

```
    // equals will use different versions of
    // bill()
    if (specialPrice.equalDeals(regularPrice))
      System.out.println("Deals are equal.");
    else
      System.out.println("Deals are not equal.");
  }
}
```

Output

```
floor mat Price and total cost = $ 10.00
floor mat Price $ 11.00 Discount = 10.0%
Total cost = $9.90
Discounted item is cheaper.
cup holder Price and total cost = $ 9.90
cup holder Price $ 11.00 Discount = 10.0%
Total cost = $9.90
Deals are equal.
```

`toString()` and Late Binding

- If you write:

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

- it will print

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

- Java uses late binding . The `println` statement above is equivalent to:

```
System.out.println
            (theObject.toString());
```

- Where it waits until it knows what the object's type is before doing the binding.

Upcasting

- ***Upcasting*** is when a program type casts an object of a derived class to a base class. (The object is now higher on the hierarchy; hence the name.)
- Example:

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

This will print:

```
Paint Price = $15.00 Discount = 10.0%
Total cose = $13.50
```

Downcasting

- ***Downcasting*** is when a program type casts an object of a base class to a derived class. (The object is now lower on the hierarchy; hence the name.)
- Example:

```
Sale saleVariable = new Sale("paint", 15);
DiscountSale discountVariable;
discountVariable = (DiscountSale)saleVariable;
```

  This will produce a runtime error. It should because there is no instance variable discount.
- Although downcasting is dangerous there are rare occasions when we will need it.

`clone()`

- Every object has a method called clone() that enables it to return a copy of the calling object.
- It does the same basic job as a copy constructor, but will work in cases where copy constructors do not do exactly what you want.

`clone()` for `Sale` and `DiscountSale` classes

```
// clone() - We'll need this to correctly
//           copy Sale objects
 public Sale clone() {
   return new Sale(this);
 }

// clone() - We'll need this to correctly
//           copy DiscountSale objects
  public DiscountSale clone() {
    return new DiscountSale(this);
  }
```

`CopyingDemo.java`

```
// Demonstrates where the clone mehod works,
// but copy constructors do not.
public class CopyingDemo  {
  public static void main(String[] args) {
    Sale[] a = new Sale[2];
    a[0] = new Sale("atomic coffee mug", 130.00);
    a[1] = new DiscountSale("invisible paint",
                                      5.00, 10);
    int i;

    Sale[] b = badCopy(a);

    System.out.println
            ("With copy constructors:");
```

```
    for (i = 0;  i < a.length;  i++) {
      System.out.println("a[" + i + "] = "
                                      + a[i]);
      System.out.println("b[" + i + "] = "
                                      + b[i]);
      System.out.println();
    }
    System.out.println();

    b = goodCopy(a);
    System.out.println("With clone method:");
    for (i = 0;  i < a.length;  i++) {
      System.out.println
             ("a[" + i + "] = "  + a[i]);
      System.out.println
                ("b[" + i + "] = "  + b[i]);
      System.out.println();
    }
  }
```

```
  public static Sale[] badCopy(Sale[] a) {
    Sale[]  b = new Sale[a.length];
    for (int i = 0; i < a.length; i++)
      b[i] = new Sale(a[i]); // Problem here!!
    return b;
  }


  public static Sale[] goodCopy(Sale[] a) {
    Sale[]  b = new Sale[a.length];
    for (int i = 0; i < a.length; i++)
      b[i] = a[i].clone();
    return b;
  }
}
```

Output from **`CopyingDemo`**

```
With copy constructors:
a[0] = atomic coffee mug Price and total cost = $ 130.00
b[0] = atomic coffee mug Price and total cost = $ 130.00

a[1] = invisible paint Price $ 5.00 Discount = 10.0%
Total cost = $4.50
b[1] = invisible paint Price and total cost = $ 5.00


With clone method:
a[0] = atomic coffee mug Price and total cost = $ 130.00
b[0] = atomic coffee mug Price and total cost = $ 130.00

a[1] = invisible paint Price $ 5.00 Discount = 10.0%
Total cost = $4.50
b[1] = invisible paint Price $ 5.00 Discount = 10.0%
Total cost = $4.50
```

Abstract Methods

- Sometimes it is necessary for a method to use another method that will be redefined in a derived class. Sometimes, it may even be difficult to define the method in the base class.
- An abstract method serves as a placeholder for a method that will be fully and more usefully defined in a derived (or descendent) class.
- An abstract method has a complete method heading that includes the modifier **`abstract`** but no method body.

Abstract Class

- An abstract class is a class with one or more abstract methods.
- An abstract class must use the modifer **`abstract`** in the class header.
- You cannot create instances of abstract classes; what you ***CAN*** do is create instances of classes that are derived from abstract classes.

Employee.java

```
import java.util.Scanner;

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

  // The abstract method
  public abstract double getPay();

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

  … …
}
```

HourlyEmployee.java

```
// Class invariant: All employees have a name,
//                  hire date
//                  non-negative wage rate and a
//                  non-negative number of ours
//                  worked.
//                  A name string of "No name"
//                  indicates
//                  no real name specified yet.
public class HourlyEmployee extends Employee {
  private double wageRate;
  private double hours;  // for the month
```

```
  // HourlyEmployee() - A default constructor
  public HourlyEmployee() {
    super();  // Call the base class's
              // constructor
    wageRate = 0;
    hours = 0;
  }

  // HourlyEmployee() - A conversion constructor
  // Precondition: Neither theName nor theDate is
  //               null;  theWageRate and
  //               theHours are non-negative.
  public HourlyEmployee(String theName,
            Date theDate, double theWageRate,
                                double theHours) {
```

```
  // using the base class's conversion
  // constructor

  super(theName, theDate);
  if ((theWageRate >= 0) && (theHours >= 0)) {
    wageRate = theWageRate;
    hours = theHours;
  }
  else {
    System.out.println("Fatal error:"+
      " creating an illegal hourly employee.");
    System.exit(0);
  }
}
```

```
// HourlyEmployee() - Copy constructor
public HourlyEmployee
     (HourlyEmployee originalObject)  {

  // the base class's copy constructor
  super(originalObject);
  wageRate = originalObject.wageRate;
  hours = originalObject.hours;
}

// The accessor of the derived class
public double getRate() {
  return wageRate;
}
```

```
public double getHours() {
  return hours;
}

// getPay() - this accessor will be different
//            in the other derived classes
public double getPay() {
  return wageRate * hours;
}
```

```
// setHours() - A mutator for hours
// Precondition: hoursWorked is non-negative
public void setHours(double hoursWorked)  {
  if (hoursWorked >= 0 )
    hours = hoursWorked;
  else {
    System.out.println
        ("Fatal error: Negative hours worked");
    System.exit(0);
  }
}
```

```
// setRate() - A mutator for wageRate
// Precondition: newWageRate is non-negative
public void setRate(double newWageRate)  {
  if (newWageRate >= 0 )
    wageRate = newWageRate;
  else {
    System.out.println
       ("Fatal error: Negative wage rate");
    System.exit(0);
  }
}
```

```
// toString() - overwrites the base class's
//              toString method
public String toString() {
  return (getName() + " "
            + getHireDate().toString()
            + "\n" + wageRate + " per hour for"
            + hours
            + " hours.");
}
```

```
  //equals() - Overrides the base class's equals
  //           method
  public boolean equals(HourlyEmployee other)  {
    return(getName().equals(other.getName()) &&
        getHireDate().equals(other.getHireDate())
             && wageRate == other.wageRate
             && hours == other.hours);
  }
}
```

EmployeeDemo.java

```
public class EmployeeDemo  {
  public static void main(String[] args)  {
    HourlyEmployee joe
        = new HourlyEmployee("Joseph Smith",
          new Date("January", 1, 2003),
                              5.50, 300);
    SalariedEmployee sam
        = new SalariedEmployee("Samuel Adams",
          new Date("March", 15, 2004), 100000);

    if (joe.samePay(sam))
      System.out.println
           (joe.getName() + " and "
            + sam.getName() + " made the same.");
    else
      System.out.println
        (joe.getName() + " and " + sam.getName()
                    + " did not make the same.");
  }
}
```