POLYMORPHISM AND ABSTRACT CLASS

CIS*2430 (Fall 2010)

Polymorphism

- Three main programming mechanisms for 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* or *static 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*
- Java uses late binding for all methods (except private, `final`, and static methods)

The Sale Class

- 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 Class

- 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

- 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`**

Sale and DiscountSale Classes

Sale class:

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

Sale class:

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

DiscountSale class:

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

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
```

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");
}
```

Pitfall with Static Methods

- No late-binding for static methods:

```
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new
    DiscountSale("floor mat", 11.00, 10);
simple = discount;
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());
}
```

- 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

- *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

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, since in many cases, it 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: Check before Downcasting

- 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`***

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

The clone Method

- The heading for the **`clone`** method defined in the **`Object`** class is as follows:

```
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 descendent class of the class **`Object`**
  - This is an example of a covariant return type

The clone Method

- If a class has a copy constructor, the **`clone`** method for that class can use the *copy constructor* to create the copy returned by the **`clone`** method

```
public Sale clone()
{
  return new Sale(this);
}
```

and another example:

```
public DiscountSale clone()
{
  return new DiscountSale(this);
}
```

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

```
b[i] = new Sale(a[i]); //plain Sale object
```

Limitations of Copy Constructors

- However, if the **`clone`** method is used instead of the copy constructor, then (because of late binding) a true copy is made, even from objects of a derived class (e.g., **`DiscountSale`**):

```
b[i] = (a[i].clone());//DiscountSale 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

Problems with Base Classes

- To take advantage of inheritance, we add the following method to the **`Employee`** class:

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

- Problem? **`samePay`** invokes **`getPay,`** which is defined in the derived classes, but not in the base **`Employee`** class and there is no way to define it reasonably without knowing whether the employee is hourly or salaried

A Solution

- 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
- Java allows this by using abstract classes and methods

Abstract Methods & 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 Methods

- 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 Classes

- 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 Classes

- 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*

No Objects for Abstract Classes

- 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**

Tip: Abstract Classes are Types

- 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
- Multiple inheritance is not allowed in Java, but can be approximated through interfaces
- 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 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

Display 13.1 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.

Implementing an Interface (1/2)

Display 13.2 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.

Implementing an Interface (2/2)

Display 13.2 Implementation of an Interface (continued)

```
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*

Abstract Classes Implementing Interfaces

Display 13.3 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

Display 13.4 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.

The Comparable Interface

- The `Comparable` interface is in the `java.lang` package, and so is automatically available to any program

- It has only the following method heading that must be implemented:
  - `public int compareTo(Object other);`

- It is the programmer's responsibility to follow the semantics of the `Comparable` interface when implementing it

The Comparable Interface Semantics

- The method `compareTo` must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other

- If the parameter `other` is not of the same type as the class being defined, then a `ClassCastException` should be thrown

GeneralizedSelectionSort (1/3)

Display 13.5 **Sorting Method for Array of Comparable** ***(Part 1 of 2)***

```
1  public class GeneralizedSelectionSort
2  {
3      /**
4       Precondition: numberUsed <= a.length;
5                     The first numberUsed indexed variables have values.
6       Action: Sorts a so that a[0, a[1], ... , a[numberUsed - 1] are in
7       increasing order by the compareTo method.
8      */
9      public static void sort(Comparable[] a, int numberUsed)
10     {
11         int index, indexOfNextSmallest;
12         for (index = 0; index < numberUsed - 1; index++)
13         {//Place the correct value in a[index]:
14             indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
15             interchange(index,indexOfNextSmallest, a);
16              //a[0], a[1],..., a[index] are correctly ordered and these are
17              //the smallest of the original array elements. The remaining
18              //positions contain the rest of the original array elements.
19         }
20     }
```

GeneralizedSelectionSort (2/3)

Display 13.5 Sorting Method for Array of Comparable *(Part 1 of 2)* (continued)

```
21      /**
22       Returns the index of the smallest value among
23       a[startIndex], a[startIndex+1], ... a[numberUsed – 1]
24      */
25      private static int indexOfSmallest(int startIndex,
26                                         Comparable[] a, int numberUsed)
27      {
28          Comparable min = a[startIndex];
29          int indexOfMin = startIndex;
30          int index;
31          for (index = startIndex + 1; index < numberUsed; index++)
32              if (a[index].compareTo(min) < 0)//if a[index] is less than min
33              {
34                  min = a[index];
35                  indexOfMin = index;
36                  //min is smallest of a[startIndex] through a[index]
37              }
38          return indexOfMin;
39      }
```

GeneralizedSelectionSort (3/3)

Display 13.5 **Sorting Method for Array of** Comparable ***(Part 2 of 2)***

```
    /**
     Precondition: i and j are legal indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, Comparable[] a)
    {
        Comparable temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }

}
```

Sorting Arrays of Comparable (1/3)

Display 13.6 **Sorting Arrays of Comparable** ***(Part 1 of 2)***

```
1  /**
2   Demonstrates sorting arrays for classes that
3   implement the Comparable interface.
4  */
5  public class ComparableDemo
6  {
7      public static void main(String[] args)
8      {
9          Double[] d = new Double[10];
10         for (int i = 0; i < d.length; i++)
11             d[i] = new Double(d.length - i);

12         System.out.println("Before sorting:");
13         int i;
14         for (i = 0; i < d.length; i++)
15             System.out.print(d[i].doubleValue() + ", ");
16         System.out.println();

17         GeneralizedSelectionSort.sort(d, d.length);

18         System.out.println("After sorting:");
19         for (i = 0; i < d.length; i++)
20             System.out.print(d[i].doubleValue() + ", ");
21         System.out.println();
```

The classes `Double` and `String` do implement the `Comparable` interface.

Sorting Arrays of Comparable (2/3)

Display 13.6 **Sorting Arrays of Comparable** ***(Part 2 of 2)***

```
22          String[] a = new String[10];
23          a[0] = "dog";
24          a[1] = "cat";
25          a[2] = "cornish game hen";
26          int numberUsed = 3;

27          System.out.println("Before sorting:");
28          for (i = 0; i < numberUsed; i++)
29              System.out.print(a[i] + ", ");
30          System.out.println();
31
32          GeneralizedSelectionSort.sort(a, numberUsed);
```

Sorting Arrays of Comparable (3/3)

Display 13.6 **Sorting Arrays of Comparable** ***(Part 2 of 2)*** (continued)

```
33            System.out.println("After sorting:");
34            for (i = 0; i < numberUsed; i++)
35                System.out.print(a[i] + ", ");
36            System.out.println();
37        }
38    }
```

SAMPLE DIALOGUE

```
Before Sorting
10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0,
After sorting:
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
Before sorting;
dog, cat, cornish game hen,
After sorting:
cat, cornish game hen, dog,
```

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

Pitfall: Inconsistent Interfaces

- Since a class may implement any number of interfaces, there are inconsistencies that can arise
  - Two interfaces may have constants with the same name, but with different values
  - Two 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