

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

- 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 `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`), overridden `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

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

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)

```
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 = (DiscountSale)saleVariable; //will produce 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
```

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
 - 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`

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:
`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

A First Look at 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);  
}
```

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

Pitfall: Sometime the `clone` Method Return Type is `Object`

- Prior to Java version 5.0, the `clone` method for the `Sale` class would have looked like this:

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

- Therefore, the result must always be type cast when using a `clone` method written for an older version of Java

```
Sale copy = (Sale)original.clone();
```


Pitfall: Sometime 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:

```
Sale copy = original.clone();
```

- However, adding the following type cast produces no problems:

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

Pitfall: 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