Lecture - Polymorphism and Abstract Classes

Polymorphism

There are four main programming mechanisms that constitute object-oriented programming (OOP)

- Abstraction
- 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*.

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 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("tyre gauge", 9.95);
System.out.println(aSale);
```

Output produced:

```
tyre gauge price and total cost = $9.95
```

This works because of late binding.

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.

Another example:

```
HourlyEmployee joe = new HourlyEmployee("Joe Worker", new Date("January", 1, 2004),
                                        50.50, 160);
Employee mike = new Employee("Mike Jordan", new Date("March", 1, 1984));

System.out.println();
System.out.println("joe's record is as follows:");
System.out.println(joe);

Sstem.out.println();
System.out.println("mike's record is as follows:");
System.out.println(mike);
```

Exercise: Write suitable `toString` functions for `HourlyEmployee` and `Employee`.

Why is this called 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*.

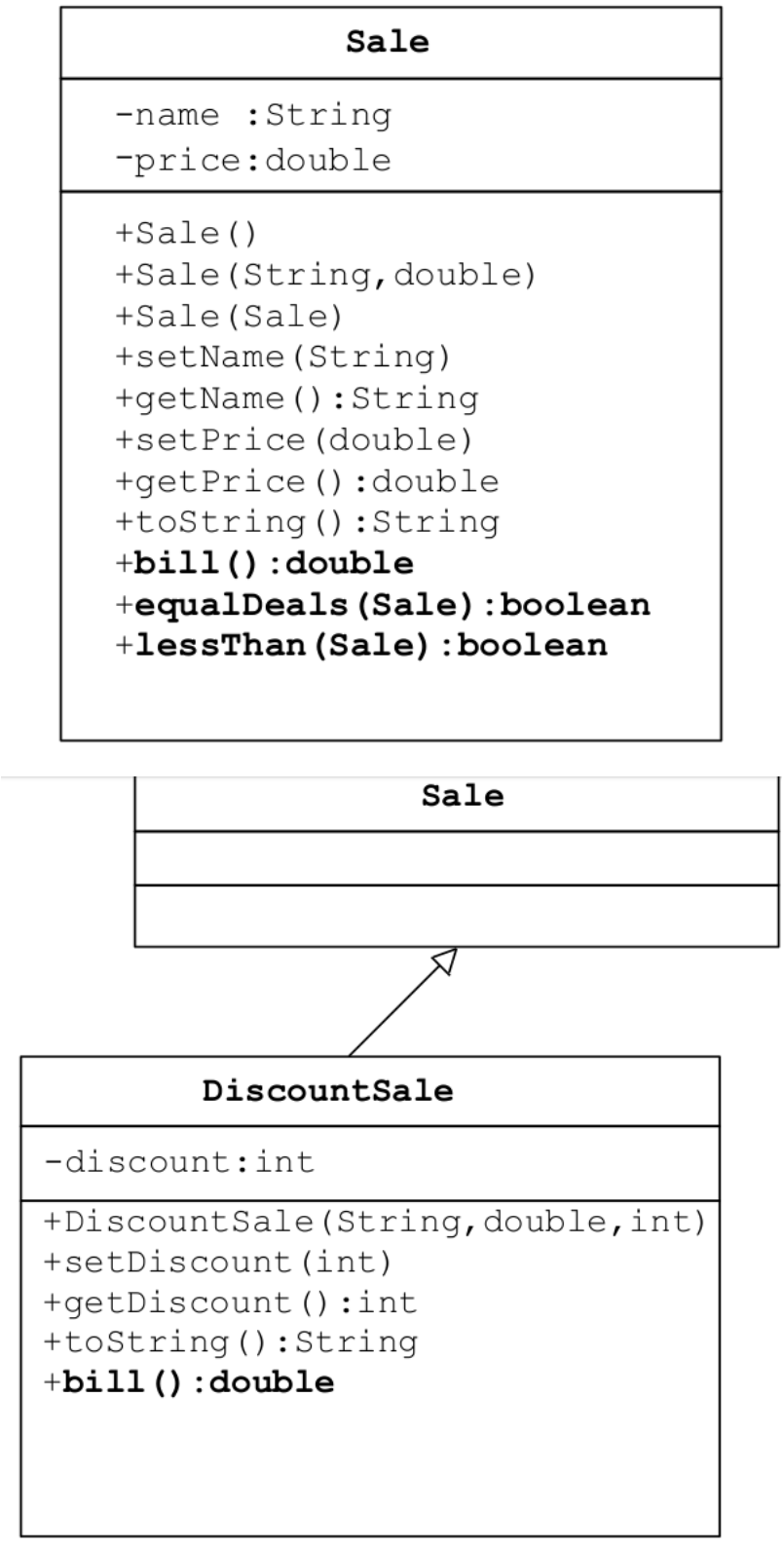
If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*.

Java uses late binding for all methods (except `private`, `final`, and `static` methods) .

Exercise: Why are those three exceptions?

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. Consider the following two classes:



The Sale class lessThan method is the following. (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 class bill() method is

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

The DiscountSale class bill() method is

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

The following

```
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!");
. . .
```

will produce the output

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

In this 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.

Advanced: Late binding is achieved by each object having a table of references to methods. The name of the method corresponds to the index into the table. The value of the entry corresponds to the appropriate method for that object to call when that method name is used. This process is an example of *indirection*, common in lower level languages.

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 still uses the definition given in the `DiscountSale` class.

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:

```
//will produce compiler error
discountVariable = saleVariable

//will produce run-time error
discountVariable = (DiscountSale)saleVariable;
```

There are times, however, when downcasting is necessary, e.g., inside the `equals` method for a class:

```
Sale otherSale = (Sale)otherObject;    //downcasting
```

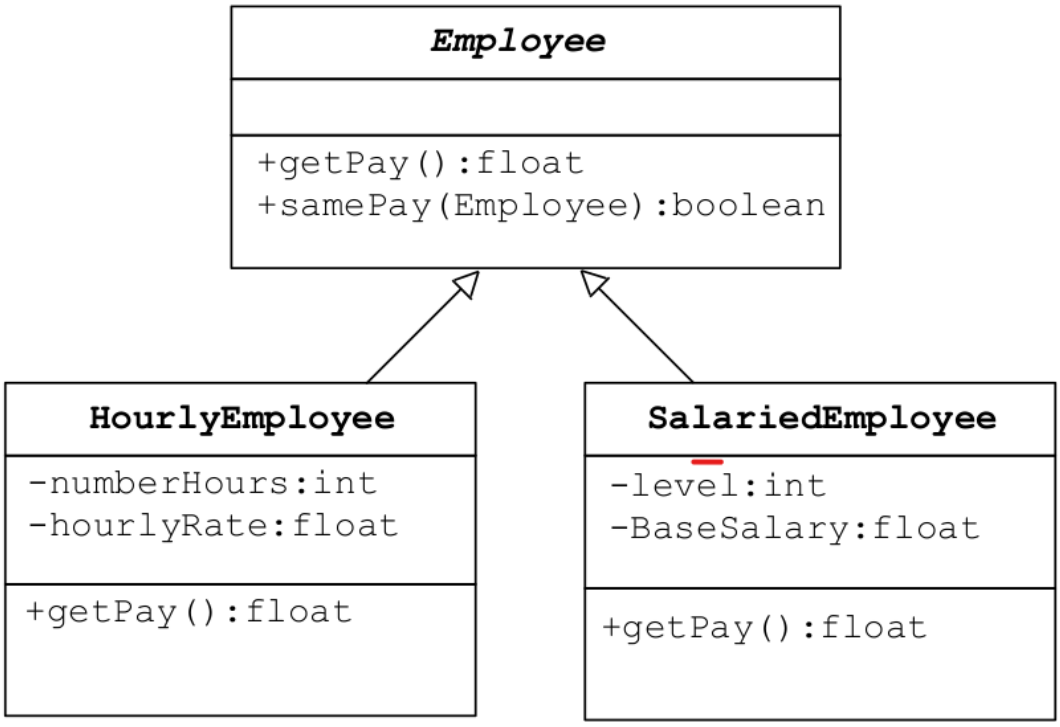
We also saw downcasting, protected by `instanceof`, used in the previous slide.

It is the esponsibility 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.

Abstract classes

In Chapter 7 of the textbook, the Employee base class and two of its derived classes, `HourlyEmployee` and `SalariedEmployee` were defined.



Imagine 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());
}
```

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.

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 using "abstract" classes and methods.

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.*

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. (**Exercise**: Why not?)

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);
```

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();
. . .
}
```

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 an instance 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`.

The constructor in an abstract class is only used by the constructor of its derived classes.

Exercise: Any abstract class can be replaced by a class that simply has methods with an empty method body replacing all virtual methods. What is the advantage of using an abstract class over the class just described? (Hint: the answer relates to this pitfall.)

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 descendant classes.

It is also fine to use a variable of an abstract class type, as long as it names objects of its concrete descendant classes only.

Round things

Create an abstract class RoundThing, with an abstract method area() and a constructor that reads parameters from a Scanner.

Create concrete derived classes Circle that defines area() properly.

Create an abstract derived class Round3DThing, with an abstract method volume().

Create concrete derived classes Sphere and Cylinder that define area() and volume() appropriately.

Write a test program that allows a user to enter two shapes and say which has the smaller area. If both are Round3DThings, say which has the smaller volume.

This is the class diagram of a sample design: