Ch 9

Inheritance and Polymorphism

Superclass and Subclass

Inheritance is the mechanism whereby a new class, called a subclass, is created from an existing class called a superclass. The subclass absorbes the state and behavior of the superclass.

We say that the subclass inherits characteristics of its superclass.

The new class inherits all the instance variables and methods (except for the constructors).

The superclass is also known as the base class or Parent class

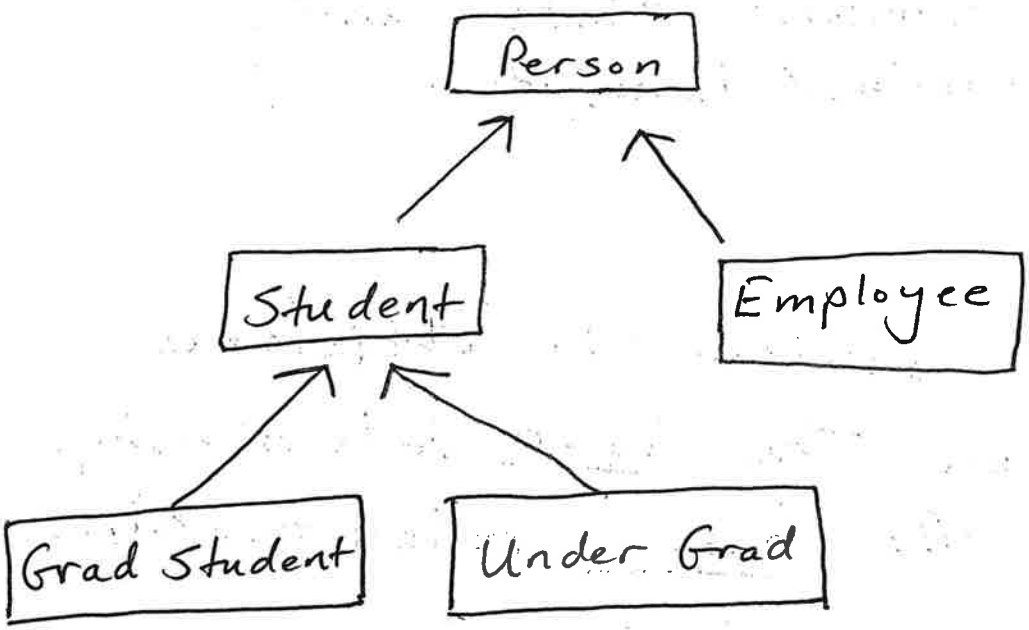
The subclass is also known as the derived class or child class.

Don't get confused by the names: a subclass is bigger than a superclass - it contains more variables and more methods.

Inheritance provides an effective mechanism for code reuse. Since a subclass object shares features of a superclass object, the only new code required is for the additional characteristics of the subclass.

A subclass can itself be a superclass for another subclass leading to an inheritance hierarchy.

Exp)



For any of these classes, an arrow points to its superclass. The arrow designates the is-a relationship.

Thus, an employee is-a person, a grad student is-a student, etc...

Note that the is-a relationship is transitive. If a GradStudent is-a Student and a Student is a Person then a GradStudent is also a Person.

(Pass out student class and variable/method handout)

Let's review the difference between the is-a relationship and the has-a relationship.

An is-a relationship relates to inheritance as we see in the student class. A has-a relationship occurs when one class has an instance variable object of another class. For example, the Employee class that you wrote has a Date object instance variable.

Implementing Subclasses and the extends keyword

The relationship between a subclass and a superclass is specified in the declaration of the subclass, using the keyword extends.

The general format looks like this:

over

public class Superclass {

- private instance variables
- other data members
- constructors
- public methods
- private methods

}

public class Subclass extends Superclass {

- additional private instance variables
- additional data members
- constructors (not inherited!)
- additional public methods
- inherited public methods whose implementation is overridden
- additional private methods

}

Inheriting Instance Methods and Variables

The UnderGrad and GradStudent subclasses inherit all of the methods and variables of the Student superclass.

Since Student instance variables myName, myTests, and myGrade are private they are not directly accessible to the methods in the UnderGrad and GradStudent subclasses. However, they can directly invoke the public accessor and mutator methods of the Superclass. For example, both the UnderGrad and GradStudent use the Student class getTestAverage() and the setGrade() methods.

If the private instance variables of the Student class were protected (instead of private), then the subclasses could directly access these variables.

Also note that classes on the same level in a hierarchy diagram do not inherit anything from eachother. For example, the UnderGrad and GradStudent classes do not inherit any variables or methods from eachother. The only thing they have in common is that they are subclasses of the Student class.

Method Overriding and the Super Keyword

A method in a superclass is overridden in a subclass by defining a method with the same return type and signature (name and parameter types).

Notice that both the UnderGrad and the GradStudent classes have their own implementation of the computeGrade method. This is called method overridding

Sometimes the code for overridding a method includes a call to the Superclass method. This is called partial overriding. Typically this occurs when the subclass method wants to do what the superclass method does, plus something extra. This is achieved by using the keyword super in the implementation. For example, the computeGrade method in the GradStudent subclass.

Constructors and the super Keyword

Constructors are never inherited. If no Constructor is written for a subclass, the superclass default Constructor with no parameters is generated. If the Superclass does not have a default Constructor, but only a constructor with parameters, a compiler error will occur. If there is a default Constructor in the superclass, inherited data members will be initialized as for the superclass.

Additional instance variables in the subclass will get a default initialization (0 for primitive types and null for reference types).

A subclass constructor can be implemented with a call to the super() method, which invokes the superclass constructor. For example, the default constructor in the UnderGrad class is identical to that of the Student class. This is implemented with the statement:

```
super( );
```

The second constructor in the UnderGrad class is called with parameters that match those in the constructor of the Student superclass.

```
public UnderGrad(String name, int[] tests, String grade)
    super(name, tests, grade);
}
```

Let's look at the constructors for the UnderGrad and GradStudent class.

Note:

① If super is used in the implementation of a subclass constructor, it must be used in the first line of the constructor body.

② If no constructor is provided in a subclass, the compiler provides the following default constructor.

```
public Subclass(){
        super();
}
```

③ Be sure to provide at least one constructor when you write a subclass.

Rules for Subclasses

1) A subclass can add new private instance variables.

2) Can add new public, private, or static methods.

3) Can override inherited methods.

4) May not redefine a public method as private.

5) May not override static methods.

6) Should define its own constructors

7) Can not directly access the private variables of its superclass. It must use accessor or mutator methods.

Declaring subclass objects.

When a superclass object is declared in a client program, that reference can refer not only to an object of the superclass, but also to objects of any of its subclasses. Thus, each of the following is legal:

```
Student s = new Student();
Student g = new GradStudent();
Student u = new UnderGrad();
```

This works because a GradStudent is-a Student and an UnderGrad is-a Student. Note that since a Student is not necessarily a GradStudent nor an UnderGrad, the following declarations are not valid.

```
GradStudent g = new Student();
UnderGrad   u = new Student();
```

Let's take a look at the following declaration a little more closely.

```
Student u = new UnderGrad();
```

We would say that:

Student u

Is a reference (or a pointer) to an object of type Student. The actual object that it refers to is UnderGrad.

Student u = new UnderGrad();

Student: generic Type
u: reference
UnderGrad(): actual type

Use handout part A) for the remaining part of the notes.

Suppose you make the method calls:

```
s.setGrade("Pass");
g.setGrade("Pass");
u.setGrade("Pass");
```

The appropriate method in Student is found since GradStudent and UnderGrad both inherit the setGrade method from Student.

The following method calls, however, won't work:

```
int studentNum = s.getID();
int underGradNum = u.getID();
```

Neither Student s nor UnderGrad u inherit the getID method from the GradStudent class. Remember, a superclass does not inherit from a subclass.

Now consider the following valid method calls:

```
s.computeGrade();
u.computeGrade();
g.computeGrade();
```

Because of polymorphism, each of the correct computeGrade methods will be called.

Polymorphism

A method that has been overridden in at least one subclass is said to be polymorphic. An example is computeGrade which is redefined in both GradStudent and UnderGrad.

Polymorphism is the mechanism of selecting the appropriate method for a particular object.

See example a) for s, g, and u declarations.

Even though s, g, and u are declared as type student,

```
s.computeGrade();
g.computeGrade();
u.computeGrade();
```

will all perform the correct operations for their particular instances.

See example c)

Here an array of five student references is created, all of them initially null. Then, three of them are assigned to actual objects. Then the appropriate computeGrade method is called. The null test is necessary because some of the array references could be null.

Note:

Polymorphism applies only to overridden methods in a subclass.

In java the selection of the correct overridden method occurs during the run of the program.

Dynamic Binding (ie. Late Binding)

Making a run-time decision about which instance method to call is known as dynamic binding or late binding.

Constrast this with selecting the correct method when methods are overloaded. The compiler selects the correct overloaded method at compile time by comparing the methods' signatures. This is known as static binding or early binding.

Think of it this way: The compiler determins if a method can be called (i.e. is it legal) while the run-time environment determines how it will be called (i.e. which overridden method should be called).

See example B).

When the code is run, the computeGrade method used will depend on the type of the actual objects refers to, which in turn depends on the user input.

Using super in a Subclass

A subclass can call a method in its superclass by using super. Suppose that the superclass method then calls another method that has been overridden in the subclass. By polymorphism, the method that is executed is the one in the subclass. The computer keeps track and executes any pending statements in either method.

See example: 9.4.1 C.S. AWESOME

```
Base b = new Derived();
b.methodOne();
```

```
public class Base
{
    public void methodOne()
    {
      System.out.print("A");
      methodTwo();
    }

    public void methodTwo()
    {
      System.out.print("B");
    }
}

public class Derived extends Base
{
    public void methodOne()
    {
       super.methodOne();
       System.out.print("C");
    }

    public void methodTwo()
    {
      super.methodTwo();
      System.out.print("D");
    }
}
```

Example 9.4.1

```
Student s = new GradStudent();
```

At compile-time

At runtime.

At compile-time s is of type Student and at run-time s is of type GradStudent.

Type Compatibility

Downcasting

Consider the statements:

```
Student s = new GradStudent();
GradStudent g = new GradStudent();
```

```
int x = s.getID(); // error
int y = g.getID(); // ok
```

Both s and g represent GradStudent objects. However, s is of type Student (at compile time), and the Student class doesn't have a getID method

At compile time, only nonprivate methods of the Student class can appear to the right of the dot operator.

Type rules for polymorphic method calls

```
a.method(b);
```

Method selected by type of a at run time

Parameter b must be of correct type at compile time

- For a declaration like

```
Superclass a = new Subclass();
```

the type of a at compile time is Superclass; at run time it is Subclass.

- At compile time, method(b) must be found in the class of a, that is, in Superclass. (This is true whether the method is polymorphic or not.) If method(b) cannot be found in the class of a, you need to do an explicit cast on a to its actual type.

Don't confuse this with polymorphism: getID is not a polymorphic method. It occurs in just the GradStudent class.

The error above can be fixed with a cast.

```
int x = ((GradStudent) s).getID();
```

Note that the outer parentheses are necessary because the dot operator has a higher precedence than casting.

Downcasting means casting superclass type to a subclass type.

- For a polymorphic method, at run time the actual type of a is determined - Subclass in this example - and method(b) is selected from Subclass. This could be inherited method if there is no overridding method.

- The type of parameter b is checked at compile time. You may need to do an explicit cast to the Subclass type to make this correct.

ClassCastException

The ClassCastException is a run-time exception thrown to signal an attempt to cast an object to a class of which it is not an instance.

Examples)

①
```
Student u = new UnderGrad();
System.out.print((String)u);
```

This will cause the exception because u is not an instance of String

②

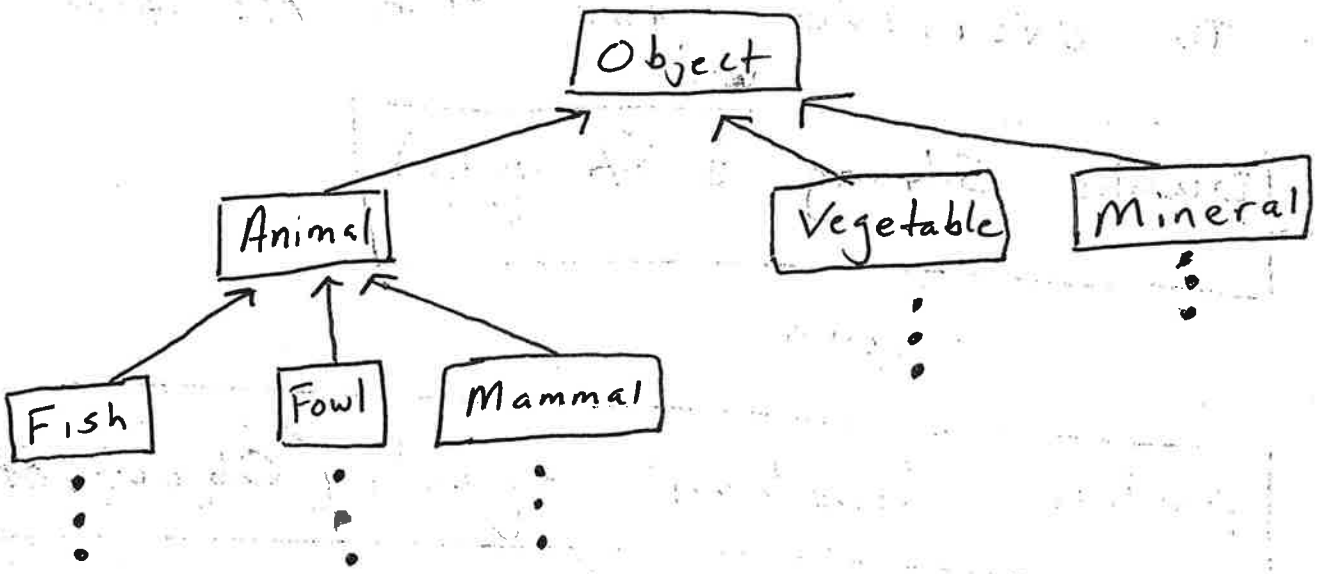
```
int x = ((GradStudent)u).getID();
```

This will cause the exception because u is not an instance of GradStudent.

The Object Class

The Universal Superclass

Every class automatically extends Object, which means that Object is a direct or indirect superclass of every other class. In a class hierarchy tree, Object is at the top.



There are many methods in Object, all of them inherited by every other class. Since Object is not an abstract class, all of its methods have implementations. The expectation is that these methods will be overridden in any class where the default implementation is not suitable. The two Object methods you need to know how to override for the AP test are

```
public String toString()
```

and

```
public boolean equals(Object other)
```

Here is an example of an equals method that compares two Points. Two points are equal if they have the same x and y values.

```
public boolean equals(Object o){
    if(o instanceof Point){
        Point other = (Point) o;
        return x == other.x && y == other.y;
    }else{
        return false;
    }
}
```

Method call:

```
p1.equals(P2);
```

The equals method returns true if this (implicit parameter) and other (explicit parameter) are the same object (reference the same memory slot)

Exp)

```
Date d1 = new Date("Jan", 14, 2001);
Date d2 = d1;
Date d3 = new Date("Jan", 14, 2001);
```

The test d1.equals(d2) returns true and d1 == d3 returns false.
d1 == d2 also returns true.

Remember not to use == to see if two objects are equal. Also, you can override the equals method to fit the needs of your programs.

The default implementation of equals is equivalent to the == relation for objects

Make sure you know how to use the following String methods (we've gone over these already) for the AP test.

1) equals

2) compareTo

3) length

4) substring (one parameter)

5) substring (two parameters)

6) indexOf

Abstraction

Abstraction may be defined as knowing as little as possible about a certain module (Class, method, or program). You only need to know what's important to you.

You don't necessarily need to know how a module works, you just need to know what input you need to provide and what output it returns.

Two types of Abstraction:

1) **Control Abstraction:**
   You don't care how a job gets done. You just know it gets done and you get a certain results.

2) **Data Abstraction:**
   In order for a method to do certain things it also needs to maintain certain pieces of data and manipulate that data as necessary.

Polymorphism

A method that has been overridden in at least one subclass is said to be polymorphic.

Polymorphism is the mechanism of selecting the appropriate method for a particular object.

Encapsulation

Combining an objects data and methods into a single unit called a class is known as encapsulation.