STUDENT LESSON

A11 – Inheritance

STUDENT LESSON

A11 – Inheritance

INTRODUCTION: Inheritance, a major component of OOP, is a technique that will allow you to define a very general class and then later define more specialized classes based upon it. You will do this by adding some new capabilities to the existing class definitions or changing the way the existing methods work. Inheritance saves work because the more specialized class inherits all the properties of the general class and you, the programmer, only need to program the new features.

The key topics for this lesson are:

A. Single Inheritance
B. Class Hierarchies
C. Using Inheritance
D. Method Overriding
E. Interfaces

VOCABULARY:

BASE CLASS
CHILD CLASS
DERIVED CLASS
`extends`
`implements`
`interface`
METHOD OVERRIDING
PARENT CLASS
SUBCLASS
`super`
SUPERCLASS

DISCUSSION:

A. Single Inheritance

1. *Inheritance* enables you to define a new class based on a class that already exists. The new class will inherit the characteristics of the existing class, but may also provide some additional capabilities. This makes programming easier, because you can reuse and extend your previous work and avoid duplication of code.

2. The class that is used as a basis for defining a new class is called a *superclass* (or *parent class* or *base class*). The new class based on the superclass is called a *subclass* (or *child class* or *derived class*.)

3. The process by which a subclass inherits characteristics from just one parent class is called single inheritance. Some languages allow a derived class to inherit from more than one parent class in a process called multiple inheritance. Multiple inheritance makes it difficult to determine which class will contribute what characteristics to the child class. Java avoids these issues by only providing support for single inheritance.

4. Figure 11.1 shows a superclass and a subclass. The line between them shows the "is a" relationship. The picture can be read as "a `Student` is a `Person`." The clouds represent the classes. That is, the picture does not show any particular `Student` or any particular `Person`, but shows that the class `Student` is a subclass of the `Person` class.

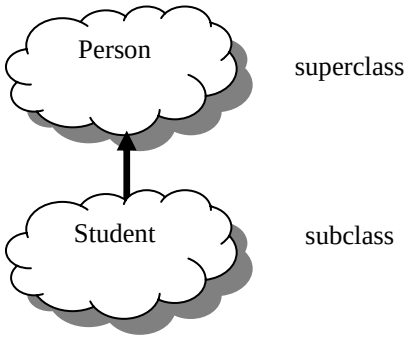

Figure 11.1 – Subclass and Superclass

5. Inheritance is between classes, not between objects. A superclass is a blueprint that is followed when a new object is constructed. That newly constructed object is another blueprint that looks much like the original, but with added features. The subclass in turn can be used to construct objects that look like the superclass's objects, but with additional capabilities.

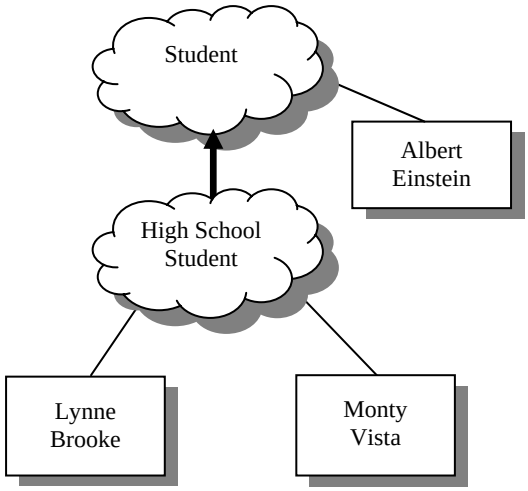

Figure 11.2 – Subclass and Superclass

6. Figure 11.2 shows a superclass and a subclass, and some objects that have been constructed from each. These objects that are shown as rectangles are actual instances of the class. In the picture, Albert Einstein, Lynne Brooke, and Monty Vista represent actual objects.

B. Class Hierarchies

1. In a hierarchy, each class has at most one superclass, but might have several subclasses. There is one class, at the top of the hierarchy that has no superclass. This is sometimes called the root of the hierarchy.

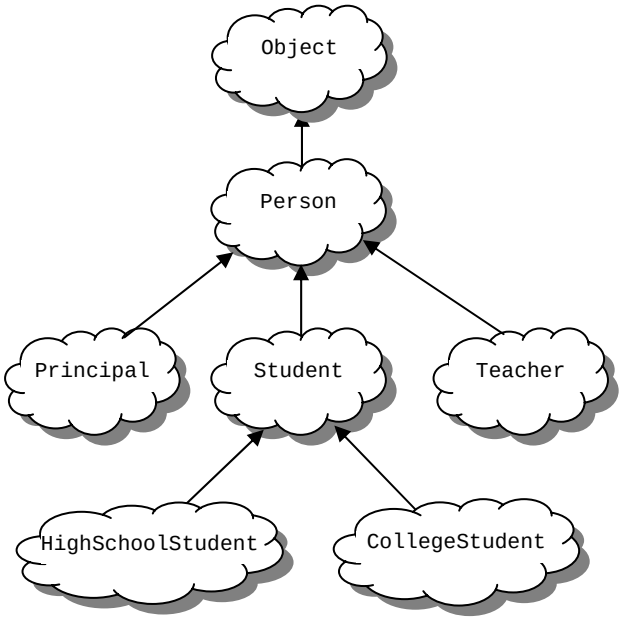

Figure 11.3 – `Person` Inheritance Hierarchy

Figure 11.3 shows a hierarchy of classes. It shows that a `Principal` is a `Person`, a `Student` is a `Person`, and that a `Teacher` is a `Person`. It also shows that both `HighSchoolStudent` and `CollegeStudent` are types of `Student`.

2. In our example, the class `Person` is the base class and the classes `Principal`, `Student`, `Teacher`, `HighSchoolStudent`, and `CollegeStudent` are derived classes.

3. In Java, the syntax for deriving a child class from a parent class is:

```
class subclass extends superclass{
  // new characteristics of the subclass go here
}
```

4. Several classes are often subclasses of the same class. A subclass may in turn become a parent class for a new subclass. This means that inheritance can extend over several "generations" of classes. This is shown in Figure 11.3, where class `HighSchoolStudent` is a subclass of class `Student,` which is itself a subclass of the `Person` class. In this case, class `HighSchoolStudent` is considered to be a subclass of the `Person` class, even though it is not a direct subclass.

5. In Java, every class that does not specifically extend another class is a subclass of the class `Object`. For example, in Figure 11.3, the `Person` class extends the class `Object`. The class `Object` has a small number of methods that make sense for all objects, such as the `toString` method, but the class `Object`'s implementations of these methods are not very useful and the implementations usually get redefined in classes lower in the hierarchy.

C. Using Inheritance

1. The following program uses a class `Person` to represent people you might find at a school. The `Person` class has basic information in it, such as name, age and gender. An additional class, `Student`, is created that is similar to `Person`, but has the Id number and grade point average of the student.

```
public class Person{
  private String myName ;   // name of the person
  private int myAge;        // person's age
  private String myGender;  // "M" for male, "F" for female

  // constructor
  public Person(String name, int age, String gender){
    myName = name;
    myAge = age;
    myGender = gender;
  }

  public String getName(){
    return myName;
  }

  public int getAge(){
    return myAge;
  }

  public String getGender(){
    return myGender;
  }

  public void setName(String name){
    myName = name;
  }

  public void setAge(int age){
    myAge = age;
  }

  public void setGender(String gender){
    myGender = gender;
  }

  public String toString(){
    return myName + ", age: " + myAge + ", gender: "
      + myGender;
  }
}

//-----------------End of Person Class-----------------//
```

```
   public class Student extends Person{
     private String myIdNum;    // Student Id Number
     private double myGPA;      // grade point average

   // constructor
   public Student(String name, int age, String gender,
                                    String idNum, double gpa){
      // use the super class' constructor
      super(name, age, gender);

      // initialize what's new to Student
      myIdNum = idNum;
      myGPA = gpa;
   }

   public String getIdNum(){
      return myIdNum;
   }

   public double getGPA(){
      return myGPA;
   }

   public void setIdNum(String idNum){
      myIdNum = idNum;
   }

   public void setGPA(double gpa){
      myGPA = gpa;
   }
}

   //-----------------End of Student Class-----------------//

public class HighSchool{
  public static void main (String args[]){
    Person bob = new Person("Coach Bob", 27, "M");
    Student lynne = new Student("Lynne Brooke", 16, "F",
                                "HS95129", 3.5);
    System.out.println(bob);
    System.out.println(lynne);
    // The previous two lines could have been written as:
    // System.out.println(bob.toString());
    // System.out.println(lynne.toString());
  }
}
```

2. The `Student` class is a derived class (subclass) of `Person`. An object of type `Student` contains `myIdNum` and `myGPA`, which are defined in `Student`. It also has indirect access to the `private` variables `myName`, `myAge`, and `myGender` from `Person` through the methods `getName()`, `getAge()`, `getGender()`, `setName()`, `setAge()`, and `setGender()` that it inherits from `Person`.

3. The constructor for the `Student` class initializes the instance data of `Student` objects and uses the `Person` class's constructor to initialize the data of the `Person` superclass. The constructor for the `Student` class looks like this:

```
// constructor
public Student(String name, int age, String gender,
               String idNum, double gpa){
  // use the super class's constructor
  super(name, age, gender);

  // initialize what's new to Student
  myIdNum = idNum;
  myGPA = gpa;
}
```

The statement **`super`**`(name, age, gender)` invokes the `Person` class's constructor to initialize the inherited data in the superclass. The next two statements initialize the members that only `Student` has. Note that when **`super`** is used in a constructor, it must be the *first* statement.

4. So far, we have only seen the **`public`** (class members that can be accessed outside the class) and **`private`** (class members that are inaccessible from outside the class) access modifiers. There is a third access modifier that can be applied to an instance variable or method. If it is declared to be **`protected`**, then it can be used in the class in which it is defined and in any subclass of that class. This declaration is less restrictive than **`private`** and more restrictive than **`public`**. The A.P. Java subset allows the use of **`protected`** with methods but discourages its use for instance variables. It is preferred that all instance variables are **`private`**. Indirect access from subclasses should be done with **`public`** "getter" and "setter" methods. While **`protected`** members are available to provide a foundation for the subclasses to build on, they are still invisible to the public at large.

D. Method Overriding

1. A derived class can *override* a method from its base class by defining a replacement method with the same signature. For example, in our `Student` subclass, the `toString()` method contained in the `Person` superclass does not reference the new variables that have been added to objects of type `Student`, so nothing new is printed out. We need a new `toString()` method in the class `Student`:

```
// overrides the toString method in the parent class
public String toString(){
  return getName() + ", age: " + getAge() + ", gender: "
         + getGender() + ", student id: " + myIdNum
         + ", gpa: " + myGPA;
}
```

A more efficient alternative is to use **`super`** to invoke the `toString()` method from the parent class while adding information unique to the `Student` subclass:

```
public String toString(){
    return super.toString() +
          ", student id: " + myIdNum + ", gpa: " + myGPA;
}
```

2. Even though the base class has a `toString()` method, the new definition of `toString()` in the derived class will override the base class's version . The base class has its method, and the derived class has its own method with the same name. With the change in the `Student` class the following program will print out the full information for both items.

```
Person bob = new Person("Coach Bob", 27, "M");
Student lynne = new Student("Lynne Brooke", 16, "F",
                            "HS95129", 3.5);

System.out.println(bob.toString());
System.out.println(lynne.toString());
```

The output to this block of code is:

```
Coach Bob, age: 27, gender: M
Lynne Brooke, age: 16, gender: F, student id: HS95129, gpa: 3.5
```

The line `bob.toString()` calls the `toString()` method defined in `Person`, and the line `lynne.toString()` calls the `toString()` method defined in `Student`.

E. Interfaces

1. In Java, an interface is a mechanism that unrelated objects use to interact with each other. Like a protocol, an interface specifies agreed-on behaviors and/or attributes.

2. The `Person` class and its class hierarchy define the attributes and behaviors of a person. But a person can interact with the world in other ways. For example, an employment program could manage a person at a school. An employment program isn't concerned with the kinds of items it handles as long as each item provides certain information, such as salary and employee ID. This interaction is enforced as a protocol of method definitions contained within an interface. The `Employable` interface would define, but not implement, methods that set and get the salary, assign an ID number, and so on.

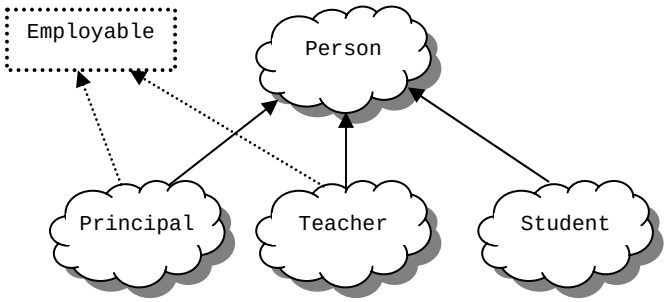

Figure 11.4 – `Employable` Interface

3. To work in the employment program, the `Teacher` class must agree to this protocol by *implementing* the interface. To implement an interface, a class must implement all of the methods and attributes defined in the interface. In our example, the shared methods of the `Employable` interface would be implemented in the `Teacher` class.

4. In Java, an **`interface`** consists of a set of methods and/or methods, without any associated implementations. Here is an example of Java interface that defines the behaviors of "employability" described earlier:

```
public interface Employable{
  public double getSalary();
  public String getEmployeeID();

  public void setSalary(double salary);
  public void setEmployeeID(String id);
}
```

   A class *implements an interface* by defining all the attributes and methods defined in the *interface*. **`implements`** is a reserved word. For example:

```
public class Teacher implements Employable{
  ...
  public double getSalary() { return mySalary; }
  public int getEmployeeID() { return myEmployeeID; }

  public void setSalary(double salary) { mySalary = salary; }
  public void setEmployeeID(String id) { myEmployeeID = id; }
}
```

5. A class can implement any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like (assuming we have an interface named `Californian`)

```
public class Teacher extends Person implements Employable,
Californian{
  ...
}
```

6. Interfaces are useful for the following:

   – Declaring a common set of methods that one or more classes are required to implement
   – Providing access to an object's programming interface without revealing the details of its class.
   – Providing a relationship between dissimilar classes without imposing an unnatural class relationship.

7. You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in future lessons.

SUMMARY/ REVIEW:

Inheritance represents the "is a" relationship between types of objects. In practice it may be used to simplify the creation of a new class. It is the primary tool for reusing your own and standard library classes. Inheritance allows a programmer to derive a new class (called a derived class or a subclass) from another class (called a base class or superclass). A derived class inherits all the data fields and methods (but not constructors) from the base class and can add its own methods or redefine some of the methods of the base class. With the size and complexity of modern programs, reusing code is the only way to write successful programs in a reasonable amount of time.

ASSIGNMENT:

Lab Assignment A11.1, *BackToSchool*
Lab Assignment A11.2, *GraphicPolygon*
Worksheet A11.1, *Inheritance Review*