[handwritten: CRITICAL IDEA IN OOP PARADIGM]

STUDENT OUTLINE

Lesson 13 – Inheritance

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

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

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 also provide some additional capabilities. This makes programming easier, because you can reuse and extend your previous work and avoid duplication of code.

[handwritten: PAIRS OF WORDS ARE COLOR CODED. DON'T MIX PAIRS!]

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*.)

[handwritten: C++ DOES THIS]

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 is 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. The figure shows a superclass and a subclass. The line between them shows the "is a kind of" relationship. The picture can be read as "a Student is a kind of 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.

"IS A KIND OF" PHRASE.
READ DIAGRAM FROM BOTTOM TO TOP.

GENERAL IDEA

Person — superclass

Student — subclass

MORE SPECIALIZED

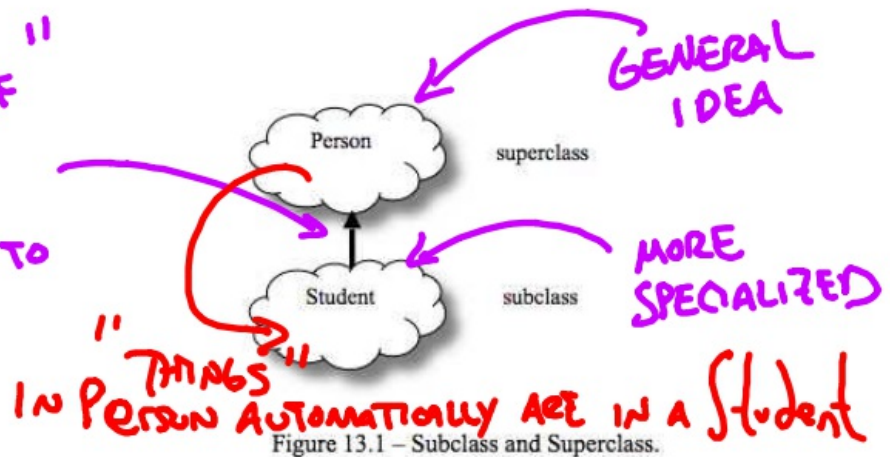**"THINGS" IN Person AUTOMATICALLY ARE IN a Student**

Figure 13.1 – 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.
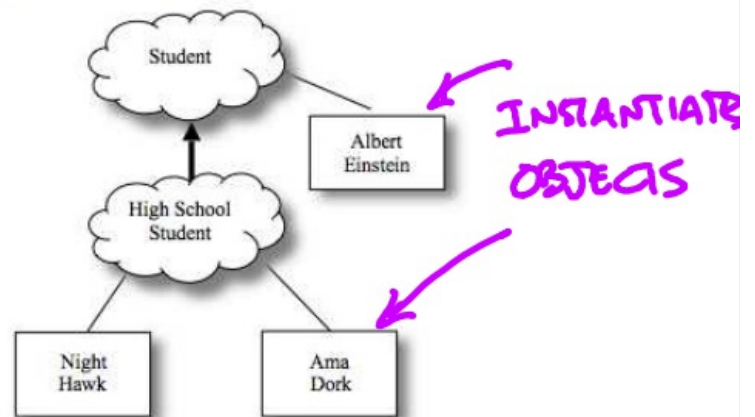
Student

Albert Einstein

High School Student

Night Hawk

Ama Dork

INSTANTIATE OBJECTS

Figure 13.2 – Subclass and Superclass.

6. The figure 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," "Night Hawk," and "Ama Dork" represent actual objects.
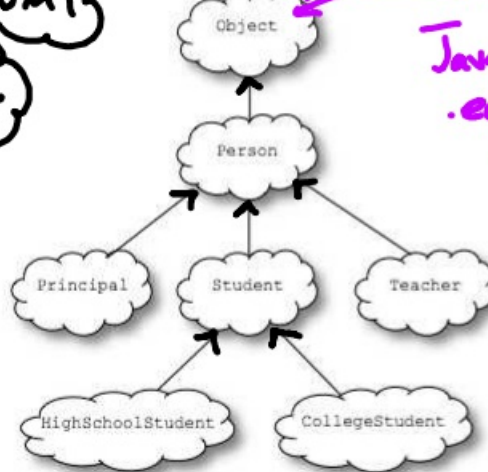
Student s1 = new Student ("Albert Einstein").
Student s2 = new HighSchoolStudent ("Ama Dork");

LEGAL??!! YES!!

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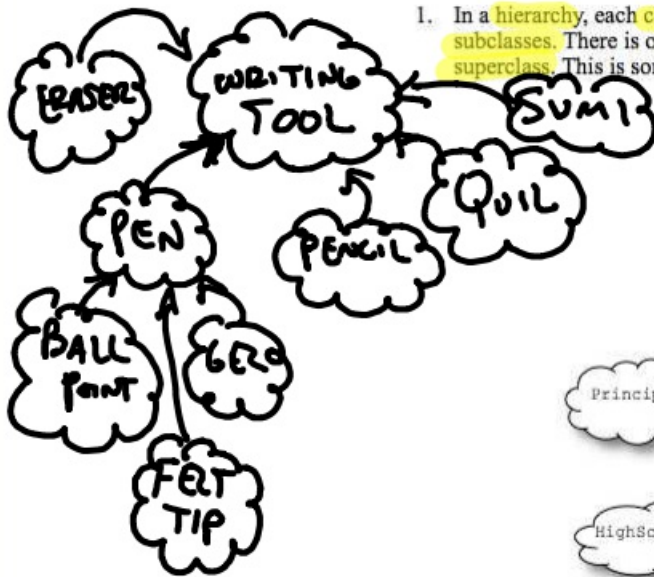
1. In a hierarchy, each class has at most one superclass, but might have several subclasses. There is one class, at the "top" of the hierarchy that has no superclass. This is sometimes called the root of the hierarchy.

[handwritten annotations: ERASER, WRITING TOOL, SUMI, PEN, PENCIL, QUIL, BALL POINT, BIRO, FELT TIP]

[handwritten: IN EVERY Java PROGRAM! .equals()]

Figure 13.3 – Person Inheritance Hierarchy.

The figure shows a hierarchy of classes. It shows that "Principal is a kind of Person," "Student is a kind of Person," and that "Teacher is a kind of Person." It also shows that "HighSchoolStudent is a kind of Student" and "CollegeStudent is a kind of Student."

[handwritten: NEW CLASS WE'RE DEFINING]

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:

[handwritten: PARENT THAT ALREADY EXISTS]

```
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 the diagram, 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 13.3, the Person class extends the Object class. The Object class has a small number of methods that make sense for all objects, such as the toString method, but these methods are not very useful and usually get redefined in classes lower in the hierarchy.

(handwritten: .equals( ))

C. Using Inheritance

1. Here is a program that 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 and grade point average of the student.

```java
class Person
{
    protected String myName ;      // name of the person
    protected int myAge;           // person's age
    protected 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 toString()
    {
        return myName + ", age: " + myAge + ", gender: " +myGender;
    }
}

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

    // 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;
    }
}

public class HighSchool
{
    public static void main (String args[])
    {
        Person bob = new Person("Coach Bob", 27, "M");
        Student del = new Student("Night Hawk", 16, "M",
                            "HS95129", 3.5);
        System.out.println(bob);
        System.out.println(del);
    }
}
```

(handwritten annotations:)
ALLOWS SUBCLASSES TO ACCESS THESE
INITIALIZE INSTANCE VARIABLES
THIS OVERRIDES THE Object VERSION OF THIS
CALLS CONSTRUCTOR OF PARENT
INHERITED FROM PERSON
INHERITS EVERYTHING IN HERE
MUST BE 1ST LINE
AUTOMATICALLY INVOKES toString()

2. The Student class is a derived class (subclass) of Person. An object of type Student contains the following members:

Member

myName	inherited from Person
myAge	inherited from Person
myGender	inherited from Person
toString()	inherited from Person
myIdNum	defined in Student
myGPA	defined in Student

(handwritten: ⓑ SEE NEXT PAGE)

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:

```java
// 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;
}
```

(handwritten: THIS MUST BE THE FIRST STATEMENT!)

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. It is not necessary to use **super**; the following would also work as a constructor for Student:

```java
// constructor
public Student(String name, int age, String gender,
               String idNum, double gpa)
{
    // initialize the inherited members
    myName = name;
    myAge = age ;
    myGender = gender;

    // initialize what's new to Student
    myIdNum = idNum;
    myGPA = gpa;
}
```
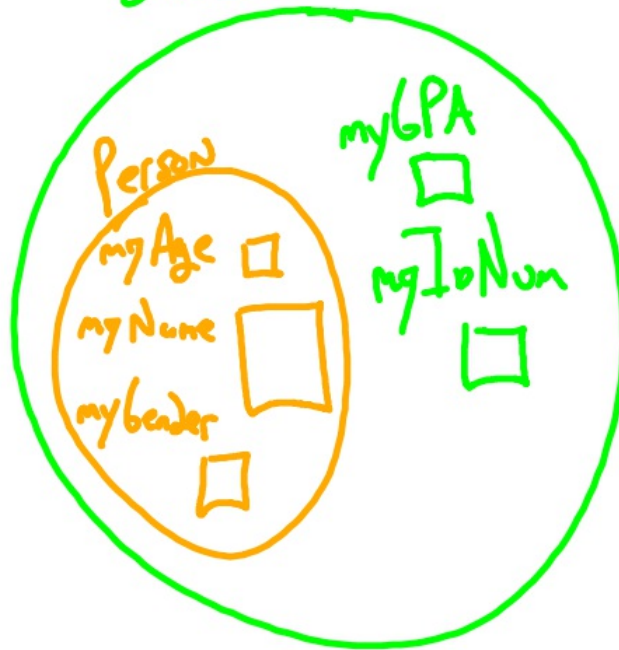
(handwritten: THIS IS DUMB. USE super..!!)

In this constructor, each variable of the newly created Student object is set to an initial value.

Student

Person

myGPA
□

myAge □

myIbNum
□

myName
□

myGender
□

5. So far, we have only seen the `private` (class members that are inaccessible from outside of the class) and `public` (class members that can be accessed outside of 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`. Classes that are written specifically to be used as a basis for making subclasses often have protected members. The protected members are there to provide a foundation for the subclasses to build on. But they are still invisible to the public at large.

D. Method Overriding

EXACT

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`:

In Student

Student SPECIFIC PARTS

```
// overrides the toString method in the parent class
public String toString()
{
   return myName + ", age: " + myAge + ", gender: " + myGender +
          ", 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.

```
class School
{
  public static void main (String args[])
  {
    Person bob = new Person("Coach Bob", 27, "M");
    Student del = new Student("Night Hawk", 16, "M",
                              "HS95129", 3.5);

    System.out.println(bob.toString());
    System.out.println(del.toString());
  }
}
```

EXPLICIT CALL (NOT REQUIRED HERE)

Run Output:

```
Coach Bob, age: 27, gender: M
Night Hawk, age: 16, gender: M, student id: HS95129, gpa: 3.5
```

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

In Student class → *(handwritten annotation)*

3. Sometimes (as in the example) you want a derived class to have its own method, but that method includes everything the derived class's method does. You can use the **super** reference in this situation to first invoke the original toString() method in the base class as follows:

INVOKES PERSON VERSION OF METHOD → *(handwritten annotation)*

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

Inside a method, **super** does not have to be used in the first statement, unlike the case when **super** is used in a constructor.

E. Interfaces

A WAY OF DOING THINGS UNIFORMLY → *(handwritten annotation)*

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

2. The Person class and its class hierarchy defines 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.

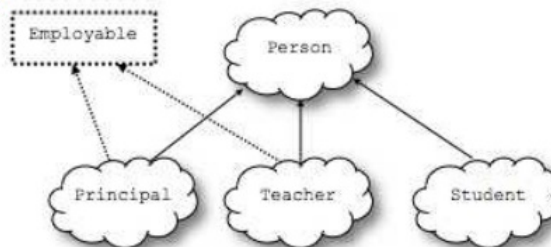*A SERIES OF METHOD SIGNATURES WITH NO IMPLEMENTATION* → *(handwritten annotation)*

Figure 13.4 – 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 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, without any associated implementations. Here is an example of Java interface that defines the behaviors of "employability" described earlier:

(handwritten: IN A STAND ALONE .java FILES)

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

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

(handwritten: ONLY SIGNATURES!!)

A class *implements an interface* by supplying code for all the interface methods. **implements** is a reserved word. For example:

(handwritten: ALL Teacher STUFF)

(handwritten: A PROMISE TO IMPLEMENT)

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

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

(handwritten: IMPLEMENT ACCESSORS)

(handwritten: IMPLEMENT MODIFIERS)

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

```java
public class Teacher extends Person implements Employable
{
    ...
}
```

(handwritten: INHERIT 1)

(handwritten: AS MANY AS YOU LIKE!)

6. An interface defines a protocol that any class anywhere in the class hierarchy can implement. 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.

(handwritten: ⊛ SEE LAST PAGE)

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 kind of" relationship between types of objects. In practice it may be used to add new features to an existing 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.
ASSIGNMENT:	Lab Exercise, L.A.13.1, *BackToSchool* Lab Exercise, L.A.13.2, *GraphicPolygon*

Animal

Invertebrates

JELLYFISH   Snail   Slug

Vertebrates

Reptile   Mammal   Avian

Snake   Gecko

Human   Dog   Penguin

Cow   Bat   Nightflad

HIERARCHIES ARE BASED ON ATTRIBUTES (GENERALLY)

CREATURES IN A ZOO SIMULATNN

Slitherable

INTERFACES ARE BASED ON ACTIONS OR BEHAVIORS

Flyable