

## Inheritance

Suppose we define a class for vehicles that has instance variables to record the vehicle's number of wheels and maximum number of occupants. The class also has accessor and mutator methods. Imagine that we then define a class for automobiles that has instance variables and methods just like the ones in the class of vehicles. In addition, our automobile class would have added instance variables for such things as the amount of fuel in the fuel tank and the license plate number and would also have some added methods. Instead of repeating the definitions of the instance variables and methods of the class of vehicles within the class of automobiles, we could use Java's inheritance mechanism, and let the automobile class inherit all the instance variables and methods of the class for vehicles.

Inheritance allows you to define a very general class and then later define more specialized classes that add some new details to the existing general class definition. This 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.

### Example:

Let's define a simple class(Base Class) called Person.

```
public class Person
{
    private String name;
    public Person()
    {
        name = "No name yet";
    }
    public Person(String initialName)
    {
        name = initialName;
    }
    public void setName(String newName)
    {
        name = newName;
    }
    public String getName()
    {
```

```

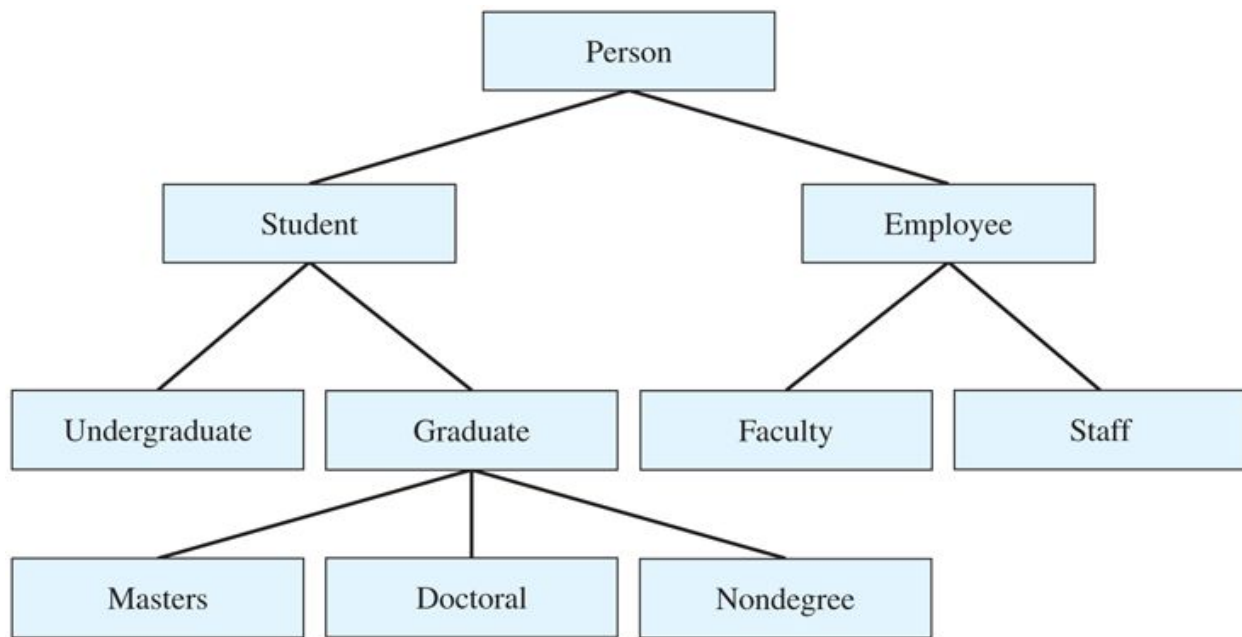
        return name;
    }
    public void writeOutput()
    {
        System.out.println("Name: " + name);
    }
    public boolean hasSameName(Person otherPerson)
    {
        return this.name.equalsIgnoreCase(otherPerson.name);
    }
}

```

## Derived Classes

Suppose we are designing a college record-keeping program that has records for students, faculty, and other staff. There is a natural hierarchy for grouping these record types: They are all records of people. **Students are one subclass of people.** Another subclass is employees, which includes both faculty and staff. Students divide into two smaller subclasses: undergraduate students and graduate students. These subclasses may further subdivide into still smaller subclasses.

## A Class Hierarchy



Example below contains the definition of a class for students. A student is a person, so we define the class Student to be a **derived class**, or **subclass**, of the class Person.

**Derived class** - A derived class is a class defined by adding instance variables and methods to an existing class. The derived class **extends** the existing class. The existing class that the derived class is built upon is called the **base class**, or **superclass**.

In our example, Person is the base class and Student is the derived class. We indicated this in the definition of Student by including the phrase extends Person on the first line of the class definition, so that the class definition of Student begins

```
public class Student extends Person
{
    private int studentNumber;
    public Student()
    {
        super();
        studentNumber = 0; //Indicating no number yet
    }
    public Student(String initialName, int initialStudentNumber)
    {
        super(initialName);
        studentNumber = initialNumber;
    }
    public void reset(String newName, int newStudentNumber)
    {
        setName(newName);
        studentNumber = newStudentNumber;
    }
    public int getStudentNumber()
    {
        return studentNumber;
    }
    public void setStudentNumber(int newStudentNumber)
    {
        studentNumber = newStudentNumber;
    }
    public void writeOutput()
    {
        System.out.println("Name: " + getName());
    }
}
```

```

        System.out.println("Student Number: " + studentNumber);
    }
    public boolean equals(Student otherStudent)
    {
        return this.hasSameName(otherStudent) &&
                (this.studentNumber ==
otherStudent.studentNumber);
    }
}

```

**Note:** A derived class extends a base class and inherits the base class's public members.

The class `Student`—like any other derived class—is said to **inherit** the public instance variables and public methods of the base class that it extends. When you define a derived class, you give only the added instance variables and the added methods. Even though the class `Student` has all the public instance variables and all the public methods of the class `Person`, we do not declare or define them in the definition of `Student`. For example, every object of the class `Student` has the method `getName`, but we do not define `getName` in the definition of the class `Student`.

A derived class, such as `Student`, can also add some instance variables or methods to those it inherits from its base class. For example, `Student` defines the instance variable `studentNumber` and the methods `reset`, `getStudentNumber`, `setStudentNumber`, `writeOutput`, and `equals`, as well as some constructors.

Notice that although `Student` does not inherit the private instance variable `name` from `Person`, it does inherit the method `setName` and all the other public methods of the base class. Thus, `Student` has indirect access to `name` and so has no need to define its own version. If `s` is a new object of the class `Student`, defined as

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


we could write

```
s.setName("Warren Peace");
```

Because `name` is a private instance variable of `Person`, however, you cannot write `s.name` outside of the definition of the class `Person`, not even within the definition of `Student`. The instance variable exists, however, and it can be accessed and changed using methods defined within `Person`.

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        Student s = new Student();
        s.setName("Warren Peace");
        s.setStudentNumber(1234);
        s.writeOutput();
    }
}
```

*setName is inherited  
from the class Person.*



### Screen Output

Name: Warren Peace  
Student Number: 1234

An object of `Student` has all of the methods of `Person` in addition to all of the methods of `Student`.

**Note:** Inheritance should define a natural is-a relationship between two classes.

If an is-a relationship does not exist between two proposed classes, do not use inheritance to derive one class from the other. Instead, consider defining an object of one class as an instance variable within the other class. That relationship is called has-a.

**A base class is often called a parent class. A derived class is then called a child class.** We can say that a child class inherits public instance variables and public methods from its parent class.

**A base class is also called a superclass, a parent class, and an ancestor class.**

You define a derived class, or subclass, by starting with another already defined class and adding (or changing) methods and instance variables. The class you start with is called the base class, or superclass. The derived class inherits all of the public methods and public instance variables from the base class and can add more instance variables and methods.

## Syntax:

```
public class Derived_Class_Name extends Base_Class_Name
{
    Declarations_of_Added_Instance_Variables
    Definitions_of_Added__And_Changed_Methods
}
```

**A derived class is also called a subclass, a child class, and a descendant class.**

## Overriding Method Definitions

The class `Student` defines a method named `writeOutput` that has no parameters. But the class `Person` also has a method by the same name that has no parameters. If the class `Student` were to inherit the method `writeOutput` from the base class `Person`, `Student` would contain two methods with the name `writeOutput`, both of which have no parameters. Java has a rule to avoid this problem. If a derived class defines a method with the same name, the same number and types of parameters, and the same return type as a method in the base class, the definition in the derived class is said to override the definition in the base class. In other words, **the definition in the derived class is the one that is used for objects of the derived class.** For example, the invocation

```
s.writeOutput();
```

will use the definition of `writeOutput` in the class `Student`, not the definition in the class `Person`, since `s` is an object of the class `Student`.

**Note: Overriding a method redefines it in a descendant class.**

When overriding a method, you can change the body of the method definition to anything you wish, but you cannot make any changes in the method's heading, including its return type.

**A method overrides another if both have the same name, return type, and parameter list.**

**So, in a derived class, if you include a method definition that has the same name, the exact same number and types of parameters, and the same return type as a**

method already in the base class, this new definition replaces the old definition of the method when objects of the derived class receive a call to the method.

When overriding a method definition, you cannot change the return type of the method. Since the signature of a method does not include the return type, you can say that when one method overrides another, both methods must have the same signature and return type.

### Overriding Versus Overloading

When you override a method definition, the new method definition given in the derived class has the same name, the same return type, and the exact same number and types of parameters. On the other hand, **if the method in the derived class were to have the same name and the same return type but a different number of parameters or a parameter of a different type from the method in the base class, the method names would be overloaded. In such cases, the derived class would have both methods.**

**Note:** A method overloads another if both have the same name and return type but different parameter lists.

**For Example** if we added the following method to the definition of the class Student,

```
public String getName(String title)
{
    return title + getName();
}
```

The class `Student` would have two methods named `getName`: It would inherit the method `getName`, with no parameters, from the base class `Person`, and it would also have the method named `getName`, with one parameter, that we just defined. This is because the two `getName` methods have different numbers of parameters, and thus the methods use overloading.

**The final Modifier:** If you want to specify that a method definition cannot be overridden by a new definition within a derived class, you can add the `final` modifier to the method heading .

For Example,

```
public final void specialMethod()
```

**Note:** A final method cannot be overridden.

**Final Class:** An entire class can be declared final, in which case you cannot use it as a base class to derive any other class. So a final class cannot be a base class.

### Private Instance Variables and Private Methods of a Base Class

An object of the derived class **Student** does not inherit the instance variable **name** from the base class **Person** , but it can access or change name's value via the public methods of Person. For example, the following statements create a Student object and set the values of the instance variables name and studentNumber:

```
public void reset(String newName, int newStudentNumber)
{
    setName(newName);
    studentNumber = newStudentNumber;
}
```

**Note:** Private instance variables in a base class are not inherited by a derived class; they cannot be referenced directly by name within a derived class.

**So the following code is INVALID:**

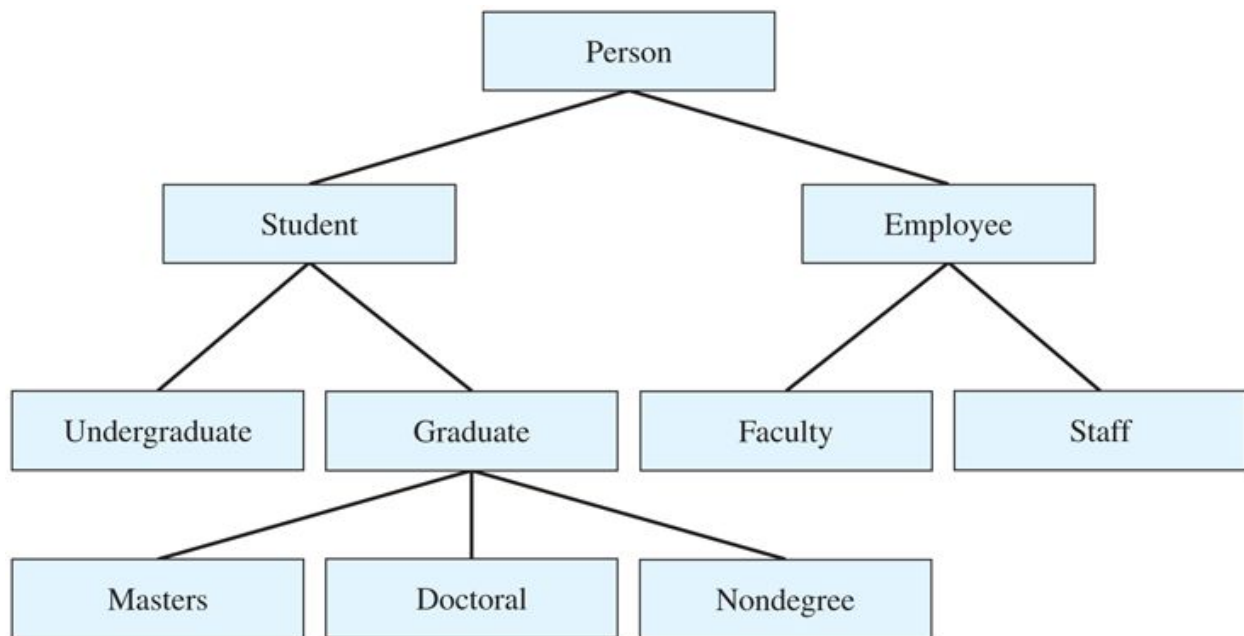
```
public void reset(String newName, int newStudentNumber)
{
    name = newName; //ILLEGAL!
    studentNumber = newStudentNumber;
}
```

A derived class cannot access the private instance variables of its base class directly by name. It knows only about the public behavior of the base class. The derived class is not supposed to know—or care—how its base class stores data. However, an inherited public method may contain a reference to a private instance variable.



## UML Inheritance Diagrams

Below is an example of class hierarchy using UML. Note that the arrowheads point up from the derived class to the base class. These arrows show the is-a relationship. For example, a **Student** is a **Person**. In Java terms, an object of type **Student** is also of type **Person**.



The arrows also help in locating method definitions. If you are looking for a method definition for some class, the arrows show the path you (or the computer) should follow. If you are looking for the definition of a method used by an object of the class **Undergraduate**, you first look in the definition of the class **Undergraduate**; if it is not there, you look in the definition of **Student**; if it is not there, you look in the definition of the class **Person**.

Look at the example below. It shows more details of the inheritance hierarchy for two classes: **Person** and one of its derived classes, **Student**. Suppose **s** references an object of the class **Student**. The diagram tells you that definition of the method **getStudentNumber** in the call

```
int num = s.getStudentNumber();
```

is found within the class `Student`, but that the definition of `setName` in

```
s.setName("Joe Student");
```

is in the class `Person`.

