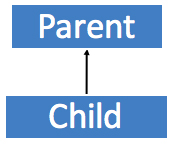
**The Shape of Inheritance**



In order to create a [class hierarchy](javascript:void(0);), you have to understand how one class can [extend](javascript:void(0);) another class. By extending classes, we can construct our class hierarchy and reduce our coding time by having classes [inherit](javascript:void(0);) methods from the classes they extend. You can compare it to how, usually, children inherit eye color from their parents. Of course, eye color inheritance is more complicated than class [inheritance,](javascript:void(0);) but it is a great example of what is occurring.

A child receives some things, by default, from the parent. A class, when extended, will pass [public](javascript:void(0);) methods and instance variables, by default, to the child class. This process is called inheritance.

Let us take a look at what is involved in actually extending a class. Extending a class means that the new class will inherit from the class it is extending. Consider an example involving rectangles and boxes. The Box class extends the Rectangle class. That means, the Box class inherits from the Rectangle class. In terms of relationship, we say that the Rectangle class is a [superclass](javascript:void(0);) of the Box class. The Box class could then also be called a [subclass](javascript:void(0);) of the Rectangle class.

This would create a class hierarchy, where Box inherits from Rectangle. Box would inherit from Rectangle both length and width. Box would extend Rectangle by adding height. This process of inheriting and extending classes is the core idea of inheritance.

The Box class will inherit any public methods that belongs to Rectangle. So if Rectangle had methods getLength and getWith, then Box will automatically have methods getLength and getWidth.

We will be able to reuse code because of this, and we will save ourselves lots of time, in terms of programming.

Suppose we were interested in creating a computer program for modeling various types of employees. We might begin by writing a class for each employee type, like these:

|  |  |
| --- | --- |
| public class SalesAssistant    {      private String myName;      private int myEmployeeID;      private int mySecureCode;       public SalesAssistant( String name,             int employeeID, int secureCode )      {        myName = name;        myEmployeeID = employeeID;        mySecureCode = secureCode;      }       public String getName()      {        return myName;      }       public int getEmployeeID()      {        return myEmployeeID;      }       public int getSecureCode()      {        return mySecureCode;      }    } | public class Teacher    {      private String myName;      private int myEmployeeID;      private int myRoomNumber;       public Teacher( String name,             int employeeID, int roomNumber )      {        myName = name;        myEmployeeID = employeeID;        myRoomNumber = roomNumber;      }       public String getName()      {        return myName;      }       public int getEmployeeID()      {        return myEmployeeID;      }       public int getRoomNumber()      {        return myRoomNumber;      }    } |

There are clear similarities between these two classes. In particular, both classes have instance variables myName and myEmployeeID, and corresponding accessor methods. There are also differences. The class representing a sales assistant has an additional field for a secure code used to open the cash register. On the other hand, the teacher has an instance variable that records the teacher's room number.

The situation we illustrated on the previous page, where several classes (SalesAssistant, Teacher, HospitalWorker, and Military) share a number of core features, is very common in Java programming. It would be very tedious, however, to have to copy all the common code from one class definition to another. To avoid the need for such time-consuming work, Java provides a way for classes to share common features without our having to include the code relating to those features within each class definition. Let us illustrate.

We begin by defining a basic Employee class that contains just the methods and instance variables that are common to all the employee classes from the previous page, like this:

public class Employee   
{   
  private String myName;   
  private int myEmployeeID;   
  
  public Employee( String name, int employeeID )   
  {   
    myName = name;   
    myEmployeeID = employeeID;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public int getEmployeeID()   
  {   
    return myEmployeeID;   
  }   
}

Having defined a class containing all the methods and instance variables shared by all our employee classes, we next define each of the more specialized classes by adding more methods and/or instance variables to the common core definition. That is, we *extend* the common core Employee class, and we indicate this relationship by employing a new keyword: extends. For example,

public class SalesAssistant extends Employee   
{   
  private int mySecureCode;   
  
  public SalesAssistant( String name, int employeeID, int secureCode )   
  {   
    super( name, employeeID );   
    mySecureCode = secureCode;   
  }   
  
  public int getSecureCode()   
  {   
    return mySecureCode;   
  }   
}

This definition informs Java that the SalesAssistant class is based on the Employee class. Note that the constructor for the SalesAssistant class takes three arguments. The first two arguments correspond to the two arguments of the Employee constructor, and the expression super( name, employeeID ) passes these arguments along to the Employee constructor.

Important note: When "extending" class definitions like this, it is important that the super statement in the extended class constructor should be the *first* statement in the constructor's body.

The statement that follows the super statement in the above definition deals with the constructor's third argument. The following code fragment shows the above code in action:

Extending classes is a key feature of OOP, which make writing programs more efficient since it promotes re-use of code. Let's look at a simple demonstration to practice this concept.

* Open the [09.01 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson01/pop/13_01b/13_01b_popa.htm).
* Create a project called 09.01 Extending Classes in the Module 09 Lessons folder.
* Download the Java files to the new project folder.
  + [Rectangle1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson01/docs/13_01b/Rectangle1.java)
  + [Box1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson01/docs/13_01b/Box1.java)
  + [ShapesTester1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson01/docs/13_01b/ShapesTester1.java)
* Read and work through the information presented in the Virtual Notes carefully.

For other practice options contact your instructor.

Working with inheritance requires great attention to detail, especially with regard to creating constructors and the use of the keyword super. Make sure you have thoroughly studied the eIMACS labs and follow the patterns established in the demonstration programs.