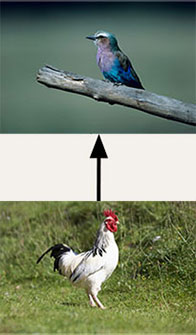
**Terrain of Hierarchies**

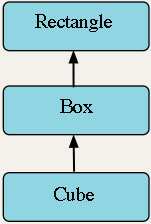


When you extend classes, you create what we call a [class hierarchy](javascript:void(0);). A class that is being extended is called the superclass, while the class doing the extending is called the subclass. The relationship can be expressed as an [is-a](javascript:void(0);) relationship. We say that the subclass is a superclass.

Consider the animal kingdom. According to classifications, a general definition of a bird has wings, a beak, and legs. Now, a rooster **is a** bird. So a rooster has wings, a beak, and legs. The bird is the superclass, while the rooster is a subclass of the bird. If we consider the flapping of the bird's wings as a public method, then a rooster inherits this public method and can flap its wings.

Whenever you extend classes, you end up with a hierarchy of classes where one class may be a [superclass](javascript:void(0);) to several subclasses. While a [subclass](javascript:void(0);) may have several superclasses, it can have only one direct superclass.

Let's examine the shapes hierarchy pictured. In this hierarchy, Rectangle is a superclass to Box and Cube. Yet Cube is not a [direct subclass](javascript:void(0);) of Rectangle.



Box is a direct superclass of Cube, and Cube is a direct subclass of Box. Rectangle is a direct superclass of Box.

Cube is a subclass of both Box and Rectangle. Another way of stating this relationship is to say that Cube is [derived](javascript:void(0);) from Box, and Box is derived from Rectangle.

Here is the definition for the SalesAssistant class:

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;   
  }   
}

When discussing the relationship between the SalesAssistant and the Employee classes, there are various phrases that are in common use. We say that

* the SalesAssistant class *extends* the Employee class;
* the Employee class is the *direct superclass* of the SalesAssistant class;
* the SalesAssistant class is a *direct subclass* of the Employee class;
* the SalesAssistant class is *derived from* the Employee class.

We also say that a class (for example, the SalesAssistant class) that is a direct subclass of another class (the Employee class) *inherits* the public methods of the direct superclass. In practical terms, this simply means that the definition of a direct subclass does not need to include copies of the definitions of the public methods provided by its direct superclass.

When we consider instances of a subclass, we sometimes refer to the ***is-a*** relationship. For example, we may say "a SalesAssistant *is an* Employee", meaning that an instance of SalesAssistant is an instance of a class that extends the Employee class. (Do not confuse this with the ***has-a*** relationship, which is sometimes used to express the fact that a class contains a particular variable, as in "an Employee *has a* myName.")

When we consider the Employee and the SalesAssistant classes, it is the SalesAssistant class that is a direct subclass of the Employee class. But it is perfectly possible for the SalesAssistant class, at the same time, to be the direct superclass of some other class. In the following code, for example, the SalesSupervisor class extends the SalesAssistant class:

public class SalesSupervisor extends SalesAssistant    
{   
  private String mySafeKey;   
  
  public SalesSupervisor( String name, int employeeID,    
                  int secureCode, String safeKey )   
  {   
    super( name, employeeID, secureCode );   
    mySafeKey = safeKey;   
  }   
  
  public String getSafeKey()   
  {   
    return mySafeKey;   
  }   
}

### public static void main( String[] args )

### {

### SalesSupervisor s = new SalesSupervisor( "Ken Blank", 18, 123, "aR62B" );

### System.out.println( "Name: " + s.getName() );

### System.out.println( "ID: " + s.getEmployeeID() );

### System.out.println( "Code: " + s.getSecureCode() );

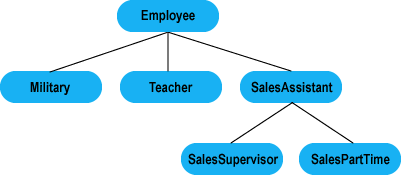
### System.out.println( "Key: " + s.getSafeKey() );

### }

Here, the SalesAssistant class is the direct superclass of the SalesSupervisor class. But the Employee class is the direct superclass of the SalesAssistant class. So what is the relationship between the Employee class and the SalesSupervisor class? We express this slightly more distant relationship by simply omitting the adjective "direct" — we say that the Employee class is a *superclass* of the SalesSupervisor class. In fact, generally speaking, class A is a superclass of class B if and only if either A is the direct superclass of B or there is some chain of direct superclass relationships that starts with A and ends with B. Similarly, we say that the SalesSupervisor class is a *subclass* of the Employee class.

Notice that the SalesSupervisor class inherits the public methods of the SalesAssistant class and the SalesAssistant class, in turn, inherits the public methods of the Employee class. So, indirectly, the SalesSupervisor class inherits the public methods of the Employee class. In general terms, a class inherits (whether directly or indirectly) all the public methods of all of its superclasses.

Over the course of the past few pages, we have defined an Employee class that we have used as the direct superclass of a number of more specialized classes, including Teacher, Military, and SalesAssisant. We have also used the SalesAssistant class as the direct superclass of the SalesSupervisor and the SalesPartTime classes, which are even more specialized. Taking them all together, we say that this collection of classes forms a *hierarchy*. This particular hierarchy may be depicted as follows:



In a hierarchy, each class that is not at the top of the tree inherits the public methods of its direct superclass (to which it is directly connected in the tree). Furthermore, it inherits the public methods of all its superclasses (direct or not). These are located along the *branch* that connects the class to the top of the tree. In the hierarchy pictured above, for example, we see that the SalesSupervisor class inherits the public methods of its direct superclass, the SalesAssistant class, and its other superclass, the Employee class. On the other hand, the Teacher class only inherits the public methods of its direct superclass, the Employee class; it has no other superclasses in this hierarchy.

We have been careful to specify that subclasses inherit *public* methods from their superclasses. Conversely, we should also make it clear that constructors are not inherited (each class, whether or not it is a subclass of another, must provide its own constructor) and nothing *private* is inherited. In particular, private instance variables are not inherited. For example, the following code — which is a modification of our earlier definition of the SalesAssistant class — generates an error. Run the code and read the error message:

public class SalesAssistant extends Employee

{

private int mySecureCode;

public SalesAssistant( String name, int employeeID, int secureCode )

{

super( name, employeeID );

mySecureCode = secureCode;

}

public boolean isTemporary()

{

// an assistant is a temp if the employeeID

// is less than 100

return (myEmployeeID < 100);

}

public int getSecureCode()

{

return mySecureCode;

}

}

public static void main( String[] args )

{

SalesAssistant s = new SalesAssistant( "Fred Brown", 18, 123 );

System.out.println( "Name: " + s.getName() );

System.out.println( "ID: " + s.getEmployeeID() );

System.out.println( "Code: " + s.getSecureCode() );

System.out.println( "Temp?: " + s.isTemporary() );

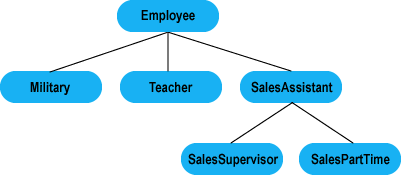
}

As the message indicates, the problem lies in the fact that the modified definition attempts to access one of the private instance variables of the Employee superclass. (Which one?) But the private marking means what it says! Those variables are not available to subclasses of the Employee class — nor indeed to anything outside the Employee class definition.

There are two ways to solve this problem. We could mark the Employee class's instance variables as public; they would then become available to any subclasses of the Employee class. Alternatively we could use the public getEmployeeID method that the SalesAssistant class inherits from its direct superclass.

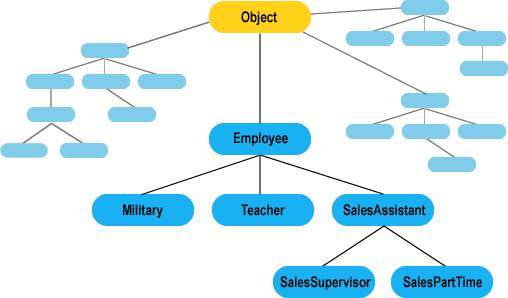
There are good reasons for preferring the second alternative. In particular, by insisting that instance variables are only accessed through the corresponding accessor methods we make it possible to track all uses of the variables with the help of a debugger (by placing a breakpoint in the accessor method, for example). This is good programming practice. Secondly, as we mentioned earlier, in the Advanced Placement Computer Science Java subset all instance variables must be marked as private.

Recall that the class hierarchy we have been discussing for the past few pages may be pictured like this:



If you examine the definitions of all these classes, you will notice that the only one whose definition does *not* involve the extends keyword is the Employee class, at the top of the hierarchy. Despite the absence of this keyword, however, the Employee class *does* have a direct superclass. Java provides a special class, named Object, that it uses as the direct superclass for classes such as Employee that do not have an explicit direct superclass. It follows that all classes inherit from Object, either directly (as Employee does) or indirectly (as SalesAssistant does).

The following illustration pictures this relationship between the Object class and the Employee hierarchy:



We show Object in a different color to emphasize that it is a class that Java provides. The illustration also includes ghostly suggestions of other class hierarchies, unrelated to the Employee hierarchy, of which Object is a superclass. There is no limit to the number of such class hierarchies — every class that we, you, or anybody else defines without an explicit direct superclass automatically has Object as its direct superclass.

There are numerous useful consequences of the fact that the Object class is a common superclass for all classes. In particular, Object has a number of public instance methods that all its subclasses inherit. We meet several of them as this course proceeds — the first, on the next page.

One of the effects of Object's status as a universal superclass is that it leads to a method for discovering the name of any class. This involves a two-stage process. One of the Object class's instance methods is getClass. When this method is invoked on any object, it returns an instance of a special class, called Class. The Class object returned by getClass represents the class of the object on which the method is invoked.

The Class class in turn provides a number of useful instance methods, among them the getName method. In the case of a Class object returned by Object's getClass method, the getName method returns a String that names the object's class. So the two-stage process of discovering the name of the class of an object is this:

* Invoke the getClass method on the object.
* Invoke the getName method on the result of the first stage.

For example, if e is a variable storing an instance of the SalesAssistant class, then the expression

e.getClass().getName()

evaluates to the String "SalesAssistant".

The Class class is unusual. It has no public constructor; instances of it are created automatically by Java as the need arises while a program is running. Furthermore, each Class object *represents* the class of an object in much the same way that your student record in your school's computer system represents you. Your student record "knows" your name, your parents' names, your address, the courses you have taken, and so on. Similarly, a Class object that represents the class of an object A "knows" the name of A's class, the name of the direct superclass of that class, and so on.

We remark in passing that, even though the getName method of the Class class and the getName method of the Employee class have the same signature, namely,

String getName()

they come from different classes and are therefore different. Which one is applied in any particular situation depends upon the data type of the object on which it is called — a Class object will use the Class method and an Employee object will use the Employee method.

In addition to getClass, the Object class provides another instance method: toString. This takes no arguments and returns a String that represents the object. Run the following program to view the string that is returned for an actual SalesAssistant object:

public static void main( String[] args )

{

SalesAssistant a = new SalesAssistant( "Ken Blank", 18, 123 );

System.out.println( a.toString() );

}

The characters that follow the "@" sign in this string give the internal name of the memory location where the SalesAssistant instance is stored. This information is often very helpful in debugging situations when we are trying to determine whether or not two instances of a class are in fact the same.

You may recall that we met something very like this [earlier](https://www.eimacs.com/eimacs/mainpage?epid=E2257478358&cid=162149#StrDoubleConv) when we were discussing converting numbers to strings and *vice versa*. There, we saw that applying Double.toString to a double or Integer.toString to an int has the effect of converting the numerical argument to the corresponding String. Clearly, each of these procedures involves a toString method. But in these cases the method takes an argument, so it cannot be the same as the Object class's toString method, which does not.

Even so, there are certain parallels in behavior. Recall that it is possible for a number to be converted *automatically* to a string, without invoking the Double.toString method or the Integer.toString method. All we have to do is concatenate the number with a string and Java takes care of the number-to-string conversion. In fact, as we pointed out at the time, the string in question could even be the empty string "". So,

"" + 5.625

evaluates to the String "5.625".

In a similar way, Java automatically converts to a String any instance of a subclass of the Object class that is involved in an expression that evaluates to a String. It does this by implicitly invoking the toString method on the object. For example, if a is a variable that stores an instance of SalesAssistant, then the expression "a is " + a behaves exactly as if it were "a is " + a.toString(), as the following code confirms:

public static void main( String[] args )

{

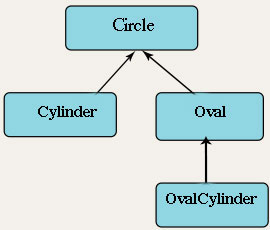
SalesAssistant a = new SalesAssistant( "Ken Blank", 18, 123 );

System.out.println( "Using toString, a is " + a.toString() );

System.out.println( "Without toString, a is " + a );

}

### Part 2



Consider the following shapes: circle, cylinder, oval, and oval cylinder. The accompanying hierarchy chart depicts the relationships among these classes.

* You may observe an oval is-a circle, a cylinder is-a circle, and an oval cylinder is-an oval.
* Oval is a super class, but Cylinder is not.

Let's explore the relationships among these classes and extend your knowledge of inheritance.

* Open the [09.02 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson02/pop/13_02b/13_02b_pop01.htm).
* Create a project called 09.02 Class Hierarchies in the Module 09 Lessons folder.
* Download the Java files to the new project folder.
  + [Circle1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson02/pop/13_02b/Circle1.java)
  + [Cylinder1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson02/pop/13_02b/Cylinder1.java)
  + [Oval1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson02/pop/13_02b/Oval1.java)
  + [OvalCylinder1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson02/pop/13_02b/OvalCylinder1.java)
  + [CircleTester1.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson02/pop/13_02b/CircleTester1.java)
* Study the discussion in the Virtual Lecture Notes carefully and explore the relational hierarchy between the super classes and the subclasses.