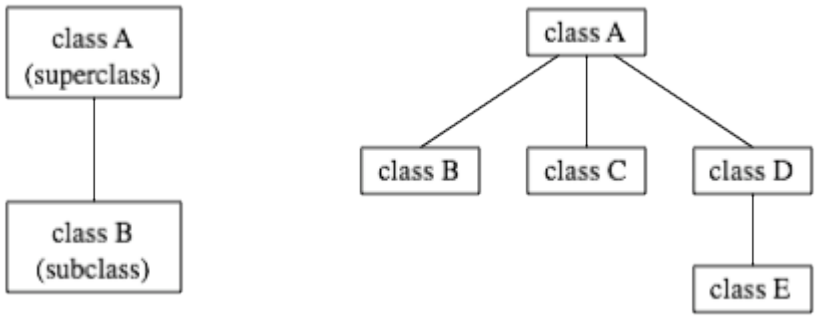
INHERITANCE AND CLASS HIERARCHY IN JAVA

The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass, as shown on the left below:



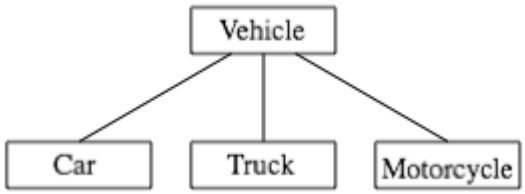
In Java, to create a class named "B" as a subclass of a class named "A", you would write

```
class B extends A {
    .
    .  // additions to, and modifications of,
    .  // stuff inherited from class A
    .
}
```

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram shown on the right above, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small class hierarchy.

5.5.3 Example: Vehicles

Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named *Vehicle* to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the *Vehicle*class, as shown in this class hierarchy diagram:



The *Vehicle* class would include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`. These are variables and methods common to all vehicles. The three subclasses of *Vehicle* -- *Car*, *Truck*, and *Motorcycle* -- could then be used to hold variables and methods specific to particular types of vehicles. The *Car* class might add an instance variable `numberOfDoors`, the *Truck* class might have `numberOfAxles`, and the *Motorcycle* class could have a boolean variable `hasSidecar`. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in a Java program would look, in outline, like this (although in practice, they would probably be `public` classes, defined in separate files):

```
class Vehicle {
   int registrationNumber;
   Person owner;   // (Assuming that a Person class has been
defined!)
   void transferOwnership(Person newOwner) {
       . . .
   }
   . . .
}

class Car extends Vehicle {
   int numberOfDoors;
   . . .
}

class Truck extends Vehicle {
   int numberOfAxles;
   . . .
}

class Motorcycle extends Vehicle {
   boolean hasSidecar;
```

```
    . . .
}
```

Suppose that `myCar` is a variable of type *Car* that has been declared and initialized with the statement

```
Car myCar = new Car();
```

Given this declaration, a program could refer to `myCar.numberOfDoors`, since `numberOfDoors` is an instance variable in the class `Car`. But since class *Car* extends class *Vehicle*, a car also has all the structure and behavior of a vehicle. This means that `myCar.registrationNumber`, `myCar.owner`, and `myCar.transferOwnership()` also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type *Car* or *Truck* or *Motorcycle* is automatically an object of type *Vehicle* too. This brings us to the following Important Fact:

A variable that can hold a reference
to an object of class A can also hold a reference
to an object belonging to any subclass of A.

The practical effect of this in our example is that an object of type *Car* can be assigned to a variable of type *Vehicle*. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable `myVehicle` holds a reference to a *Vehicle* object that happens to be an instance of the subclass, *Car*. The object

"remembers" that it is in fact a *Car*, and not **just** a*Vehicle*. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the `instanceof` operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, the assignment statement

```
myCar = myVehicle;
```

would be illegal because `myVehicle` could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously in Subsection 2.5.6: The computer will not allow you to assign an int value to a variable of type short, because not every int is a short. Similarly, it will not allow you to assign a value of type *Vehicle* to a variable of type *Car* because not every vehicle is a car. As in the case of ints and shorts, the solution here is to use type-casting. If, for some reason, you happen to know that `myVehicle` does in fact refer to a *Car*, you can use the type cast `(Car)myVehicle` to tell the computer to treat `myVehicle` as if it were actually of type *Car*. So, you could say

```
myCar = (Car)myVehicle;
```

and you could even refer to `((Car)myVehicle).numberOfDoors`. (The parentheses are necessary because of precedence. The `"."` has higher precedence than the type-cast, so`(Car)myVehicle.numberOfDoors` would try to type-cast the int `myVehicle.numberOfDoors` into a *Vehicle*, which is impossible.)

As an example of how this could be used in a program, suppose that you want to print out relevant data about the *Vehicle* referred to by `myVehicle`. If it's a *car*, you will want to print out the car's`numberOfDoors`, but you can't say `myVehicle.numberOfDoors`, since there is no `numberOfDoors` in the *Vehicle* class. But you could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                                + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c;
   c = (Car)myVehicle;  // Type-cast to get access to
numberOfDoors!
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
   Truck t;
   t = (Truck)myVehicle;  // Type-cast to get access to
numberOfAxles!
   System.out.println("Number of axles:  " + t.numberOfAxles);
}
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle:  Motorcycle");
   Motorcycle m;
   m = (Motorcycle)myVehicle;  // Type-cast to get access to
hasSidecar!
   System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if `myVehicle` refers to an object of type *Truck*, then the type cast `(Car)myVehicle`would be an error. When this happens, an exception of type *ClassCastException* is thrown. This check is done at

run time, not compile time, because the actual type of the object referred to by `myVehicle` is not known when the program is compiled.

Source : http://math.hws.edu/javanotes/c5/s5.html