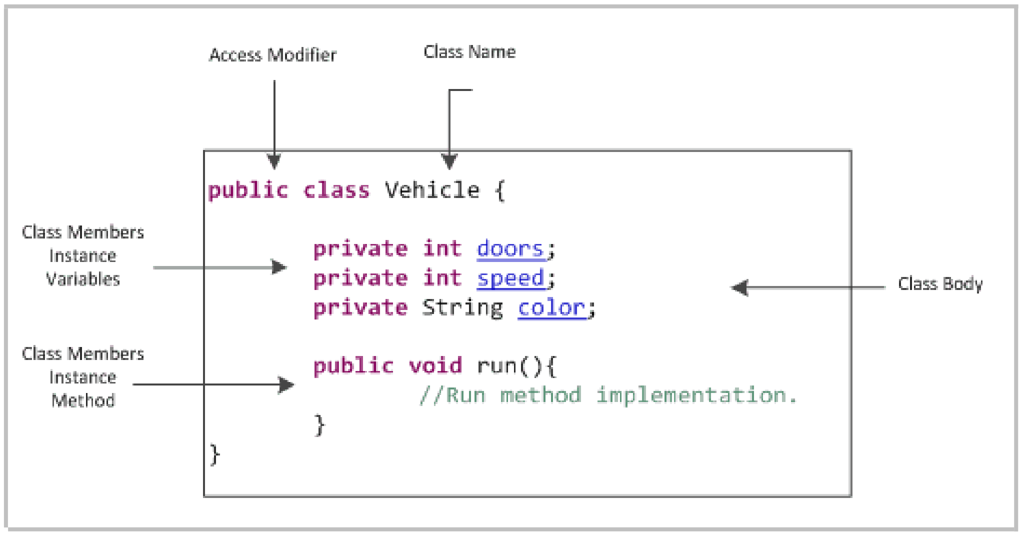
Concept of OOP

Java Declaration and Access Modifiers

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. The first way is called process-oriented model. Procedural languages such as C employ this model to considerable success. To manage increasing complexity the second approach called object-oriented programming was conceived. An object-oriented program can be characterized as data controlling access to the code. Java is object-oriented programming language. Java classes consist of variables and methods (also known as instance members). Java variables are two types either primitive types or reference types. First, let us discuss how to declare a class, variables and methods then we will discuss access modifiers.

Declaration of Class:

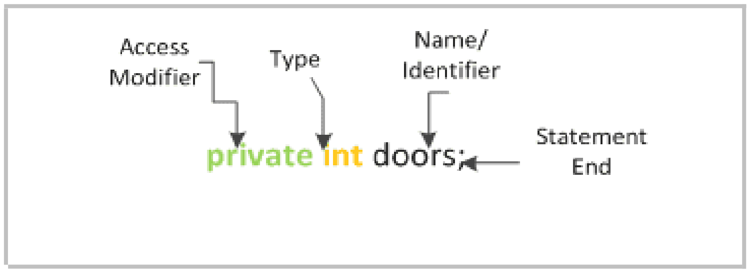
A class is declared by use of the **class** keyword. The class body is enclosed between **curly braces { and }**. The data or variables, defined within a class are called **instance variables**. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.


Access Modifier
Class Name
Class Members Instance Variables
Class Members Instance Method
Class Body

```
public class Vehicle {

        private int doors;
        private int speed;
        private String color;

        public void run(){
                //Run method implementation.
        }
}
```

Declaration of Instance Variables :

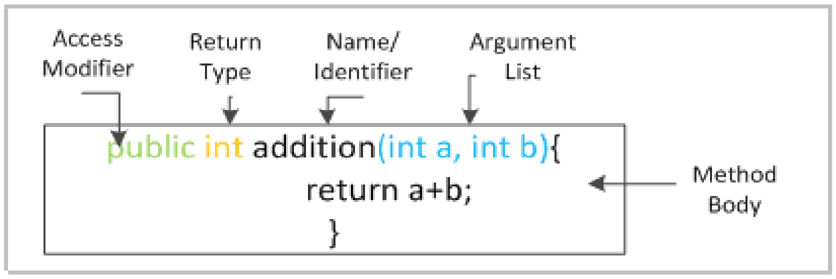
Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) **contains its own copy of these variables.** Thus, the data for one object is separate and unique from the data for another. An instance variable can be declared public or private or default (no modifier). When we do not want our variable's value to be changed out-side our class we should declare them **private**. public variables can be accessed and changed from outside of the class. We will have more information in OOP concept tutorial. The syntax is shown below.



```
private int doors;
```

Declaration of Methods :

A method is a program module that contains a series of statements that carry out a task. To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method. Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times. The syntax to declare method is given below.

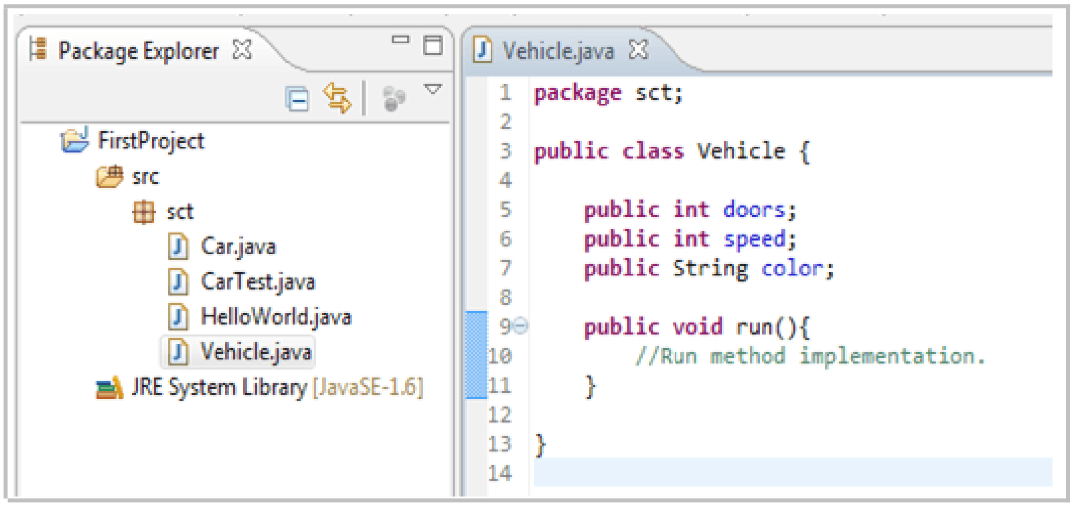


```
public int addition(int a, int b){
        return a+b;
}
```

Access modifiers:

Each object has members (members can be variable and methods) which can be declared to have specific access. Java has 4 access level and 3 access modifiers. Access levels are listed below in the least to most restrictive order.

public: Members (variables, methods, and constructors) declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package. Below screen shot shows eclipse view of public class with public members.



```
package sct;

public class Vehicle {

    public int doors;
    public int speed;
    public String color;

    public void run(){
        //Run method implementation.
    }

}

```

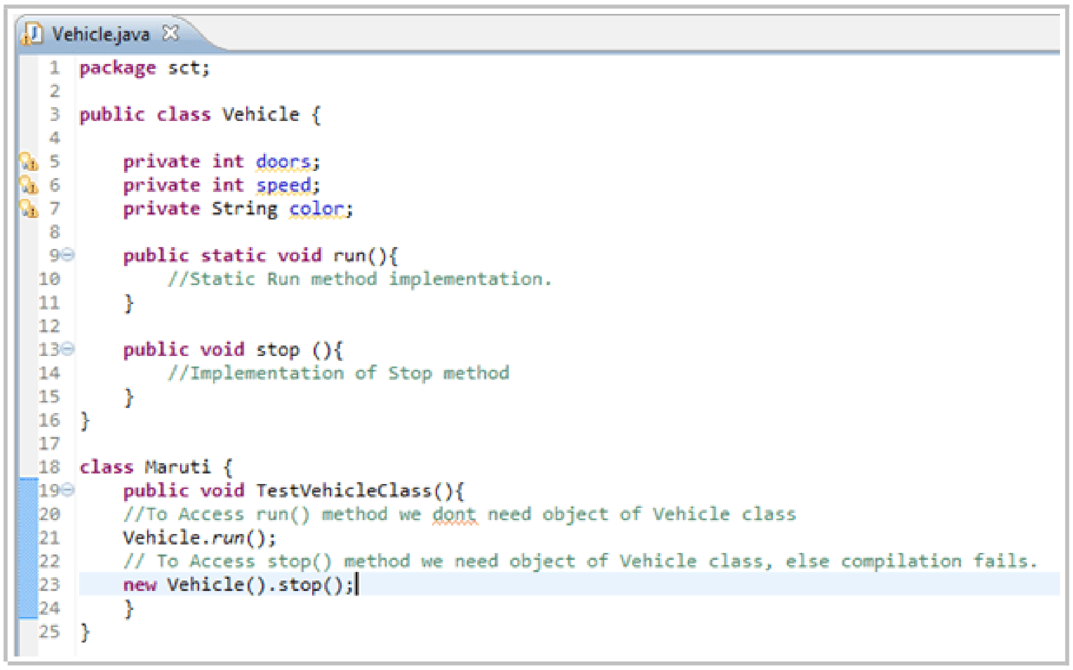
protected: The protected fields or methods, cannot be used for classes and Interfaces. Fields, methods and constructors declared protected in a super-class can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

Default (no value): The default access level is declared by not writing any access modifier at all. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package.

private: The private (most restrictive) modifiers can be used for members but cannot be used for classes and Interfaces. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accessed by anywhere outside the enclosing class.

Java has modifiers other than access modifiers listed below:

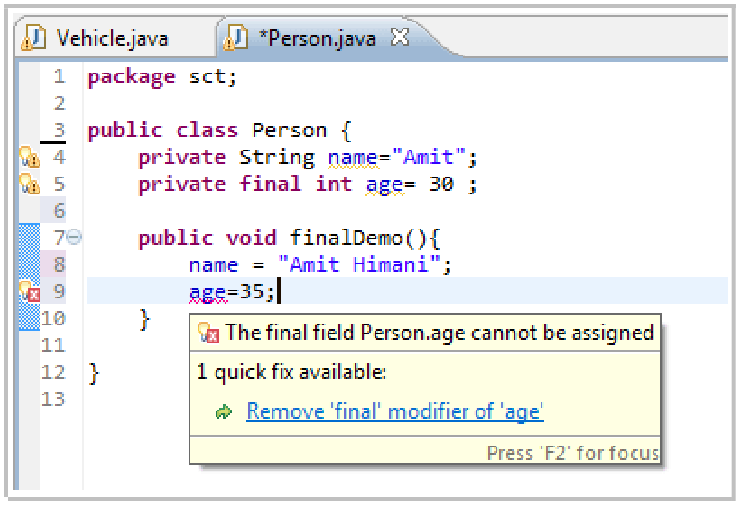
static: static can be used for members of a class. The static members of the class can be accessed without creating an object of a class. Let's take an example of Vehicle class which has run () as a static method and stop () as a non-static method. In Maruti class we can see how to access static method run () and non-static method stop ().



Vehicle.java

```java
package sct;

public class Vehicle {

    private int doors;
    private int speed;
    private String color;

    public static void run(){
        //Static Run method implementation.
    }

    public void stop (){
        //Implementation of Stop method
    }
}

class Maruti {
    public void TestVehicleClass(){
    //To Access run() method we dont need object of Vehicle class
    Vehicle.run();
    // To Access stop() method we need object of Vehicle class, else compilation fails.
    new Vehicle().stop();
    }
}
```

final: This modifier applicable to class, method, and variables. This modifier tells the compiler not to change the value of a variable once assigned. If applied to class, it cannot be sub-classed. If applied to a method, the method cannot be overridden in sub-class. In below sample, we can see compiler errors while trying to change the value of filed age because it is defined as final while we can change the value of name field.



Vehicle.java   *Person.java

```java
package sct;

public class Person {
    private String name="Amit";
    private final int age= 30 ;

    public void finalDemo(){
        name = "Amit Himani";
        age=35;
    }

}
```

The final field Person.age cannot be assigned
1 quick fix available:
Remove 'final' modifier of 'age'
Press 'F2' for focus

abstract: There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. This modifier is applicable to class and methods only. We will discuss abstract class in detail in separate Tutorial.

Below Table summarizes the access modifiers

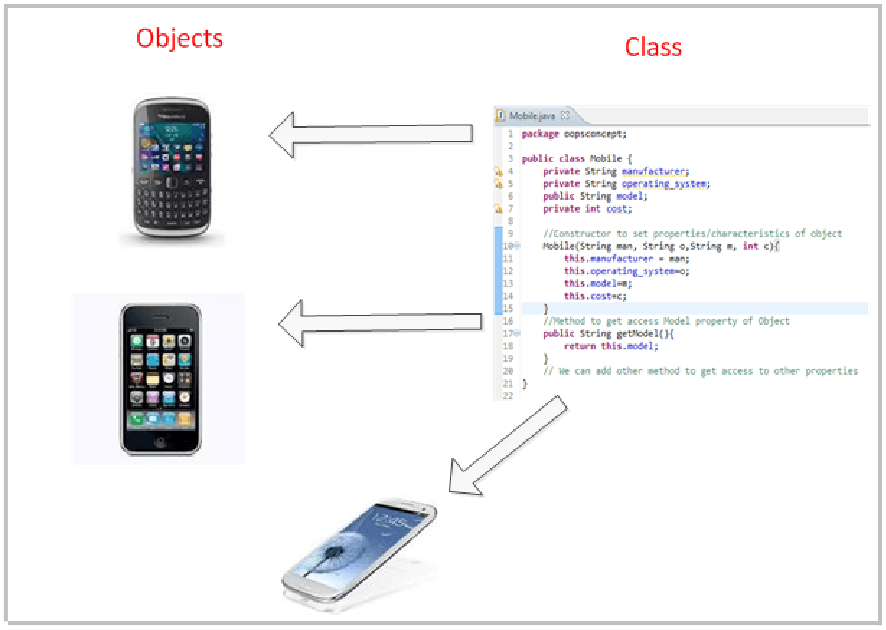
Modifier	**class**	**constructor**	**method**	**Data/variables**
public	Yes	Yes	Yes	Yes
protected		Yes	Yes	Yes
default	Yes	Yes	Yes	Yes
private		Yes	Yes	Yes
static			Yes	
final	Yes		Yes	

Let's take first column example to interpret. A "class" can have public, default, final and abstract access modifiers.

Class

This tutorial will help you to understand about Java OOP'S concepts with examples. Let's discuss what are the features of Object Oriented Programming. Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, which are stand-alone executable programs that use those objects.

A class is a template, blueprint,or contract that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. A Java class uses variables to define data fields and methods to define actions. Additionally,a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.



```java
package oopsconcept;

public class Mobile {
    private String manufacturer;
    private String operating_system;
    public String model;
    private int cost;

    //Constructor to set properties/characteristics of object
    Mobile(String man, String o,String m, int c){
        this.manufacturer = man;
        this.operating_system=o;
        this.model=m;
        this.cost=c;
    }
    //Method to get access Model property of Object
    public String getModel(){
        return this.model;
    }
    // We can add other method to get access to other properties
}
```

Objects are made up of attributes and methods. Attributes are the characteristics that define an object; the values contained in attributes differentiate objects of the same class from one another. To understand this better let's take the example of Mobile as an object. Mobile has characteristics like a model, manufacturer, cost, operating system etc. So if we create "Samsung" mobile object and "IPhone" mobile object we can distinguish them from characteristics. The values of the attributes of an object are also referred to as the object's state.

There are three main features of OOPS.

1) Encapsulation

2) Inheritance

3) Polymorphism

Let's we discuss the features in details.

Encapsulation

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. **Encapsulation provides the security that keeps data and methods safe from inadvertent changes.** Programmers sometimes refer to encapsulation as using a "black box," or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getXXX() methods which are having public access modifiers.

```
package oopsconcept;
public class Mobile {
	private String manufacturer;
	private String operating_system;
	public String model;
	private int cost;

	//Constructor to set properties/characteristics of object
	Mobile(String man, String o,String m, int c){
		this.manufacturer = man;
		this.operating_system=o;
		this.model=m;
		this.cost=c;
	}
	//Method to get access Model property of Object
	public String getModel(){
		return this.model;
	}
	// We can add other method to get access to other properties
}
```

Inheritance

An important feature of object-oriented programs is inheritance—**the ability to create classes that share the attributes and methods of existing classes, but with more specific features**. **Inheritance is mainly used for**

code reusability. So you are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it's called as Inheritance. You can look into the following example for inheritance concept. Here we have Mobile class extended by other specific class like Android and Blackberry.

```
package oopsconcept;
public class Android extends Mobile{
		//Constructor to set properties/characteristics of object
		Android(String man, String o,String m, int c){
				super(man, o, m, c);
			}
		//Method to get access Model property of Object
		public String getModel(){
			return "This is Android Mobile- " + model;
		}
}

package oopsconcept;
public class Blackberry extends Mobile{
	//Constructor to set properties/characteristics of object
	Blackberry(String man, String o,String m, int c){
					super(man, o, m, c);
				}
	public String getModel(){
		return "This is Blackberry-"+ model;
	}
}
```

Polymorphism

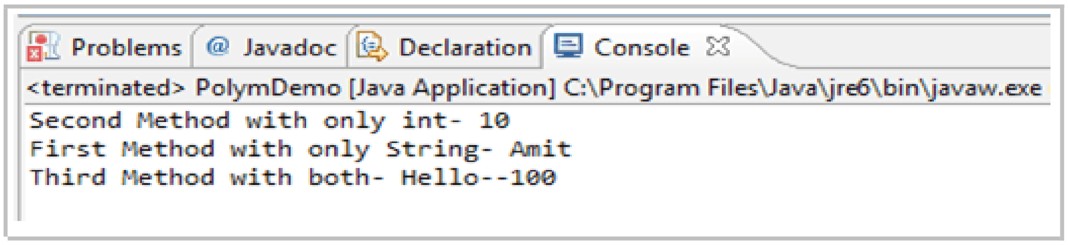
In Core, Java Polymorphism is one of easy concept to understand. Polymorphism definition is that Poly means many and morphos means forms. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. There are two types of Polymorphism available in Java. For example, in English, the verb “run” means different things if you use it with “a footrace,” a “business,” or “a computer.” You understand the meaning of “run” based on the other words used with it. Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism.

Static Polymorphism (compile time polymorphism/ Method overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading. In below program, we have three print methods each with different arguments. When you properly overload a method, you can call it providing different argument lists, and the appropriate version of the method executes.

```
package oopsconcept;
class Overloadsample {
	public void print(String s){
		System.out.println("First Method with only String- "+ s);
	}
	public void print (int i){
		System.out.println("Second Method with only int- "+ i);
	}
	public void print (String s, int i){
		System.out.println("Third Method with both- "+ s + "--" + i);
	}
}
public class PolymDemo {
	public static void main(String[] args) {
		Overloadsample obj = new Overloadsample();
		obj.print(10);
		obj.print("Amit");
		obj.print("Hello", 100);
	}
}
```

Output:



```
<terminated> PolymDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Second Method with only int- 10
First Method with only String- Amit
Third Method with both- Hello--100
```

Dynamic Polymorphism (run time polymorphism/ Method Overriding)

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members.

```
package oopsconcept;
public class OverridingDemo {
	public static void main(String[] args) {
		//Creating Object of SuperClass and calling getModel Method
		Mobile m = new Mobile("Nokia", "Win8", "Lumia",10000);
		System.out.println(m.getModel());

		//Creating Object of Sublcass and calling getModel Method
		Android a = new Android("Samsung", "Android", "Grand",30000);
		System.out.println(a.getModel());

		//Creating Object of Sublcass and calling getModel Method
		Blackberry b = new Blackberry("BlackB", "RIM", "Curve",20000);
		System.out.println(b.getModel());
	}
}
```

Abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. An essential element of object-oriented programming is an abstraction. Humans manage complexity through abstraction. When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic). What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car have different implementation of the car working but its basic interface has not changed (i.e. you still use the steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, a car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system)through the use of hierarchical abstractions.

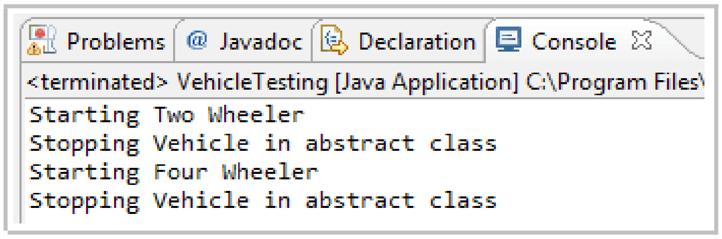
An abstract class is something which is incomplete and you can not create an instance of the abstract class. If you want to use it you need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way, Java has a concept of abstract classes, abstract method but a variable can not be abstract in Java.

Let's take an example of Java Abstract Class called Vehicle. When I am creating a class called Vehicle, I know there should be methods like start() and Stop() but don't know start and stop mechanism of every vehicle since they could have different start and stop mechanism e.g some can be started by a kick or some can be by pressing buttons.

The advantage of Abstraction is if there is a new type of vehicle introduced we might just need to add one class which extends Vehicle Abstract class and implement specific methods. The interface of start and stop method would be same.

```
class TwoWheeler extends VehicleAbstract{
	@Override
	public void start() {
		System.out.println("Starting Two Wheeler");
w	}
}
class FourWheeler extends VehicleAbstract{
	@Override
	public void start() {
		System.out.println("Starting Four Wheeler");
	}
}



package oopsconcept;
public class VehicleTesting {
        public static void main(String[] args) {
                VehicleAbstract my2Wheeler = new TwoWheeler();
                VehicleAbstract my4Wheeler = new FourWheeler();
                my2Wheeler.start();
                my2Wheeler.stop();
                my4Wheeler.start();
                my4Wheeler.stop();
        }
}
```

Output :



```
<terminated> VehicleTesting [Java Application] C:\Program Files\
Starting Two Wheeler
Stopping Vehicle in abstract class
Starting Four Wheeler
Stopping Vehicle in abstract class
```

Inheritance in Java

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Important terminology

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

What all can be done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.