Modifiers

JAVA

Access modifiers

Introduction

Do you define how people would access some of your properties? You would not want anyone using your properties. but, your close friends and relatives can have some access.. Similarly Java controls access, too.

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member.

Or

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods.

Introduction

Java provides 4 levels/ types of access modifiers.

- Private: The access level of a private modifier is **only within the class**. It cannot be accessed from outside the class.
- Default: The access level of a default modifier is **only within the package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- Protected: The access level of a protected modifier is **within the package and outside the package through child class**. If you do not make the child class, it cannot be accessed from outside the package.
- Public: The access level of a public modifier is **everywhere**. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	within class	within package by subclass	within package by non subclass	outside package by subclass only	outside package
Private	Y	N	N	N	N
Default	Y	Y	Y	N	N
Protecte d	Y	Y	Y	Y	N
Public	Y	Y	Y	Y	Y

Public access modifier

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn’t put any restriction on the access.

```
package abcpackage;

public class Addition {

   public int addTwoNumbers(int a, int b){

    return a+b;   }

}
```

- Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

```
package xyzpackage;

import abcpackage.*;

public class Test{

   public static void main(String args[])
{

    Addition obj = new Addition();

          obj.addTwoNumbers(10, 21);   }

}
```

Private access modifier

The scope of private modifier is limited to the class only.

- Private Data members and methods are only accessible within the class
- Class and Interface cannot be declared as private
- If a class has private constructor then you cannot create the object of that class from outside of the class.

```
class ABC{
   private double num = 100;
   private int square(int a){
    return a*a;
   }
}
```

```
public class Example{
   public static void main(String args[])
{
    ABC obj = new ABC();
    System.out.println(obj.num);
    System.out.println(obj.square(10));
   }
}
```

Default access modifier

- When we do not mention any access modifier, it is called default access modifier.
- The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class.
- Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.

```
package abcpackage;

public class Addition {

   int addTwoNumbers(int a, int b){

    return a+b;

   }

}
```

```
package xyzpackage;

import abcpackage.*;

public class Test {

   public static void main(String args[]){

    Addition obj = new Addition();

      /* It will throw error because we are trying to
access the default method in another package*/

    obj.addTwoNumbers(10, 21);   }

}
```

```
Exception in thread "main" java.lang.Error:
Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type
Addition is not visible
at xyzpackage.Test.main
```

Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package.

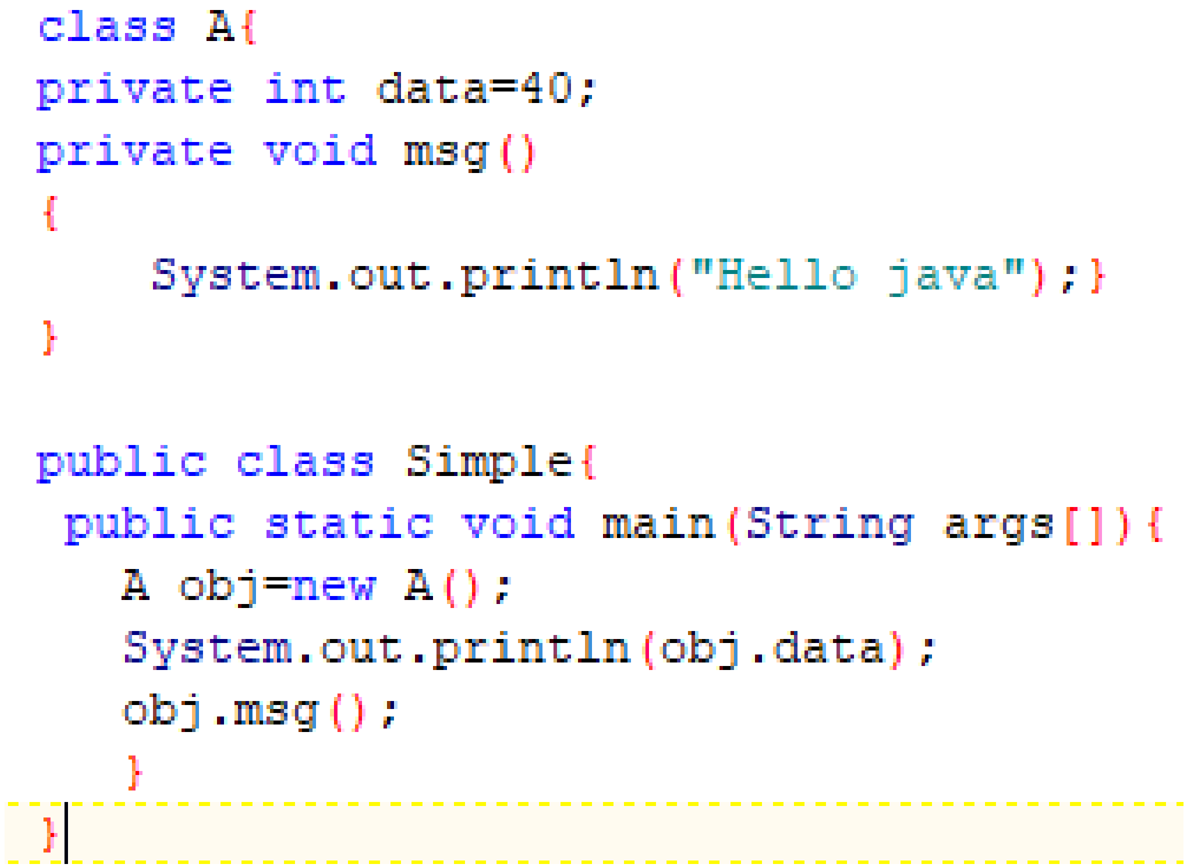
You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub-classes.

Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

```
package abcpackage;
public class Addition {
   protected int addTwoNumbers(int a, int b){
    return a+b;   }
}
```

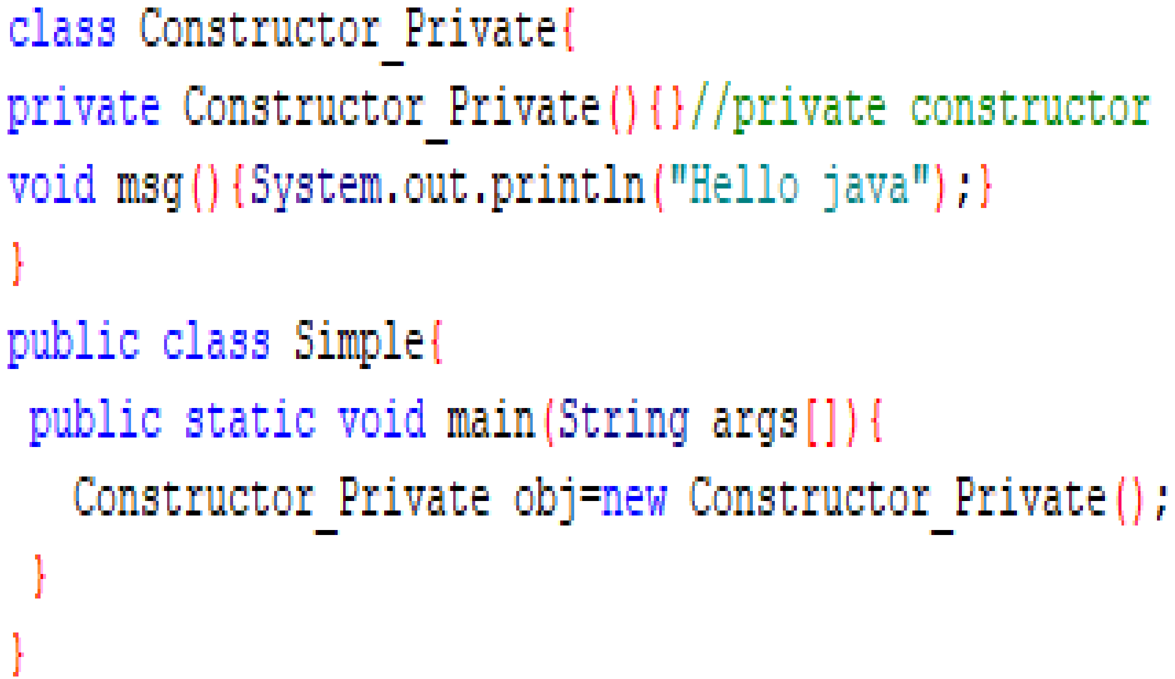
- class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected.
- This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses

```
package xyzpackage;
import abcpackage.*;
public class Test extends Addition{
   public static void main(String args[])
{
    Addition obj = new Addition();
          obj.addTwoNumbers(10, 21);   }
}
```



```java
class A{
private int data=40;
private void msg()
{
     System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);
   obj.msg();
   }
}
```

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.



```
class Constructor_Private{
private Constructor_Private(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   Constructor_Private obj=new Constructor_Private();
 }
}
```

If you make any class constructor private, you cannot create the instance of that class from outside the class.

Non Access Modifiers

Introduction

Java provides a number of non-access modifiers to achieve many other functionality.

- The **static** modifier for creating class methods and variables
- The **final** modifier for finalizing the implementations of classes, methods, and variables.
- The **abstract** modifier for creating abstract classes and methods.
- The **synchronized** and **volatile** modifiers, which are used for threads.

static Modifier

Variables:

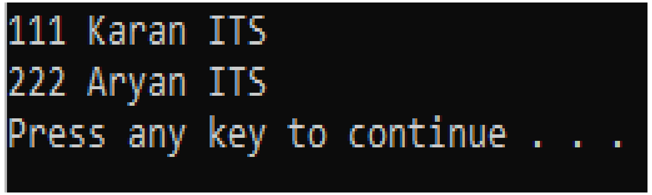
- The static keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.
- Static variables are also known as class variables. Local variables cannot be declared static.

static Modifier

Methods:

- The static keyword is used to create methods that will exist independently of any instances created for the class.
- Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.
- Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

```java
class Static_Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Static_Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable{
 public static void main(String args[]){
 Static_Student s1 = new Static_Student(111,"Karan");
 Static_Student s2 = new Static_Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Static_Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```



```
111 Karan ITS
222 Aryan ITS
Press any key to continue . . .
```

Final Modifier:

Final is the modifier applicable for classes, methods and variables.

Methods:

- Whatever the methods parent has by default available to the child.
- If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot overridden.

Class:

- If a class declared as the final then we cannot creates the child class that is inheritance concept is not applicable for final classes.

Variables:

- Final variable are similar to symbolic constants. They cannot be declared in methods.

Final Modifier:

Variables:

- A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.
- However the data within the object can be changed. So the state of the object can be changed but not the reference.
- With variables, the final modifier often is used with static to make the constant a class variable.

```
class Test{
    final int value=10; // The following are examples of declaring constants:
    public static final int BOXWIDTH = 6;
    static final String TITLE = "Manager";
    public void changeValue(){
        value = 12; //will give an error } }
```

Final Modifier:

Methods:

- A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.
- The main intention of making a method final would be that the content of the method should not be changed by any outsider.

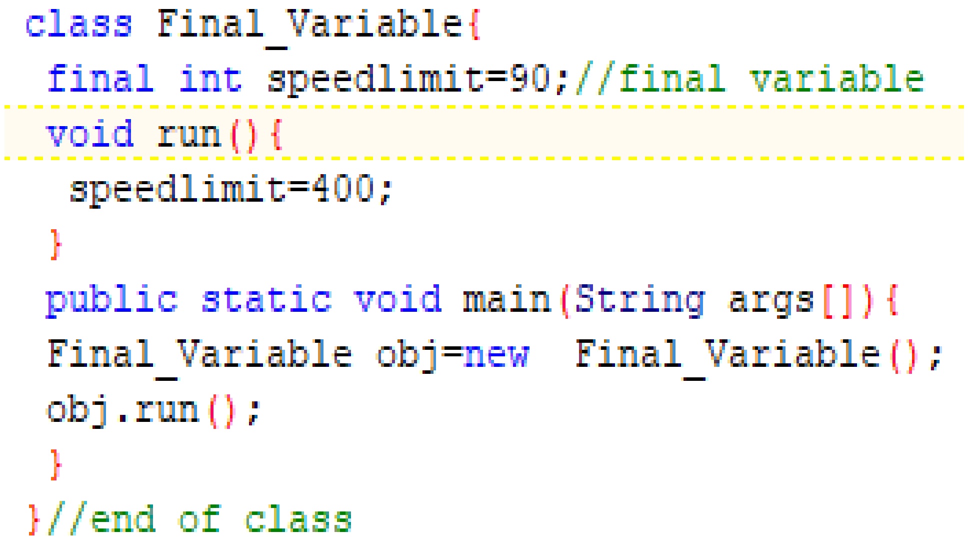
```
class Test{
        public final void changeName()
        { // body of method }
        }
```

Final Modifier:

Class:

- The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

```
public final class Test

{

    // body of class

}
```



```java
class Final_Variable{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Final_Variable obj=new  Final_Variable();
 obj.run();
 }
}//end of class
```

```java
final class Bike{}

class Final_Class extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Final_Class honda= new Final_Class();
  honda.run();
  }
}
```

abstract Modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

Methods:

- Even though we don't have implementation still we can declare a method with abstract modifier.
- That is abstract methods have only declaration but not implementation.
- Hence abstract method declaration should compulsory ends with semicolon.

Class:

- For any java class if we are not allow to create an object such type of class we have to declare with abstract modifier that is for abstract class instantiation is not possible.

abstract Modifier:

Class:

- An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.
- A class cannot be both abstract and final. (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.
- An abstract class may contain both abstract methods as well normal methods.

```
abstract class Caravan{
    private double price;
        private String model;
        private String year;
        public abstract void goFast(); //an abstract method
        public abstract void changeColor();
        }
```

abstract Modifier:

Methods:

- An abstract method is a method declared with out any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.
- Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.
- If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.
- The abstract method ends with a semicolon. Example: public abstract sample();

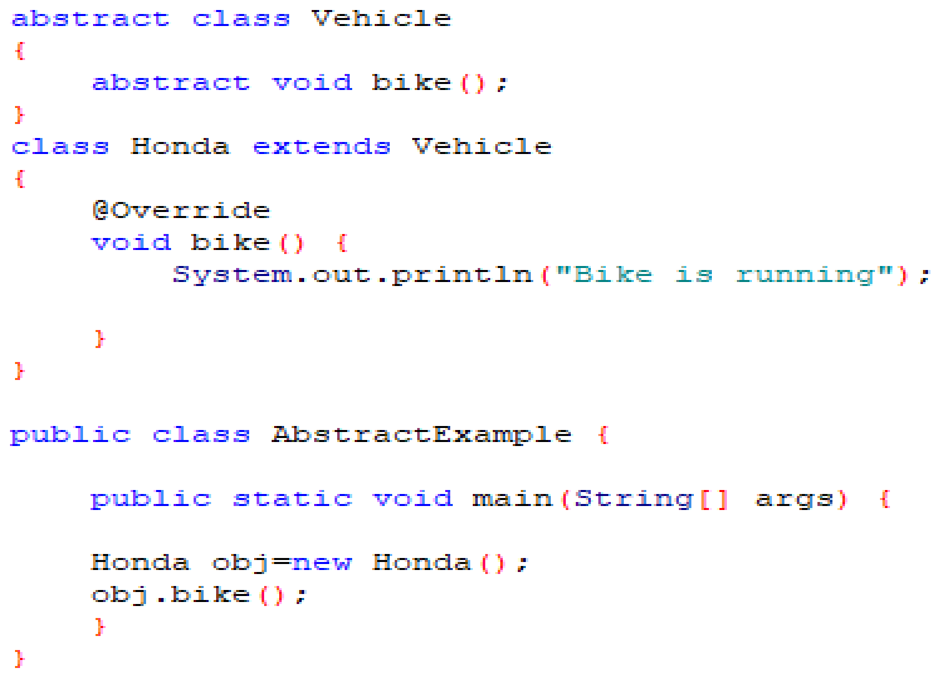
public abstract class SuperClass{

abstract void m(); //abstract method }

class SubClass extends SuperClass{ // implements the abstract method

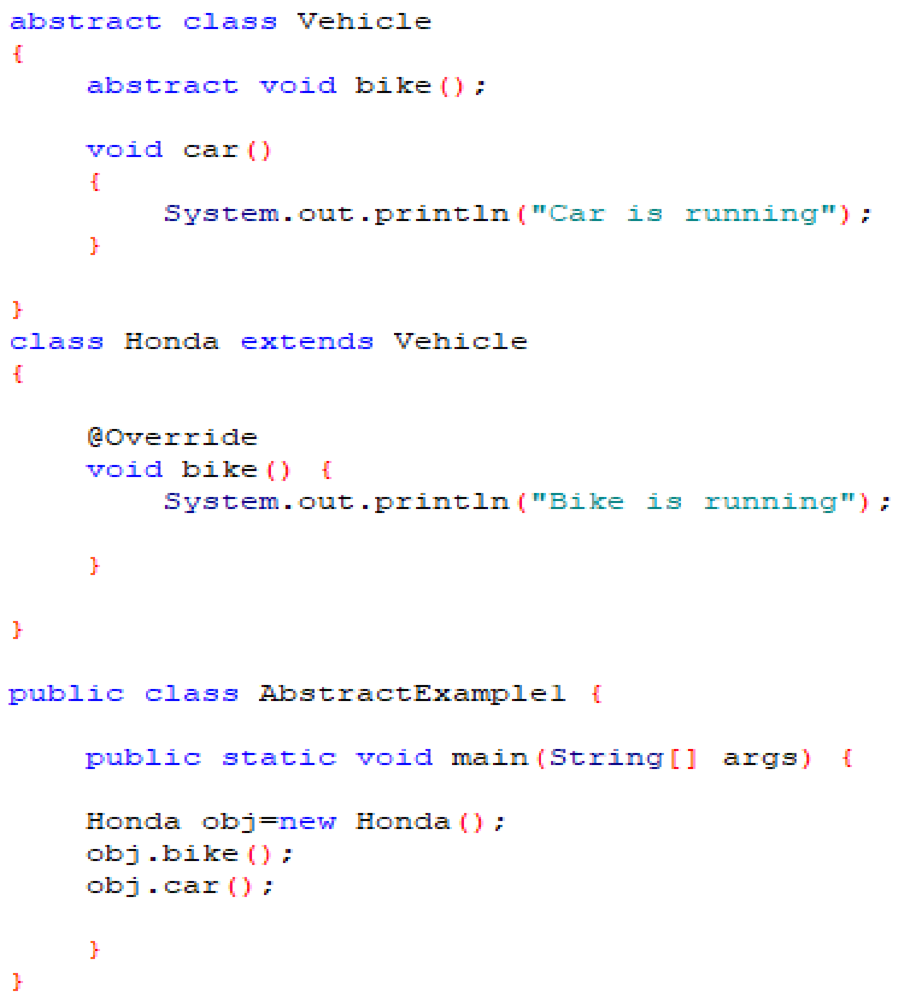
void m(){ .... } }

Abstract class containing the abstract method:



```
abstract class Vehicle
{
    abstract void bike();
}
class Honda extends Vehicle
{
    @Override
    void bike() {
        System.out.println("Bike is running");

    }
}

public class AbstractExample {

    public static void main(String[] args) {

    Honda obj=new Honda();
    obj.bike();
    }
}
```

Abstract class containing the abstract a



```
abstract class Vehicle
{
    abstract void bike();

    void car()
    {
        System.out.println("Car is running");
    }

}
class Honda extends Vehicle
{

    @Override
    void bike() {
        System.out.println("Bike is running");

    }

}

public class AbstractExample1 {

    public static void main(String[] args) {

    Honda obj=new Honda();
    obj.bike();
    obj.car();

    }
}
```

synchronized Modifier:

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

```
public synchronized void showDetails(){ ....... }
```