**Accessibility Modifiers/ Specifiers**

##### **“*Specifiers specify access and modifier modifies access.”***

**Access Modifiers:**

Access modifiers in java are used to control the visibility of a field, method, class and constructor.

**Access specifiers:**

**Access specifiers**, specify the access to our code for other classes – whether other classes can access or not and if permitted, to what extent they can access.

**Access Specifier – Meaning**

**Observe.**

***1. public void display() { }  
2. private void display() { }***

There are two **display()** methods with **public** and **private**.

These two words give permissions to other classes to access **display()** method.

**public** means any class from anywhere can access (like a public park; any one can enter or leave without ticket).

**private** means not accessible to other classes; the method can be used by the same class (in which it is declared, like a private property used by the same family for which it belongs). Now I think it is clear.

The **public**and **private** are known as access Specifiers because they specify the access.

There are four access Specifiers Java supports, **public**, **protected**, **defaul**t (not specified at all) and **private** with different access restrictions (permissions). In fact, these four are keywords also.

**B) Access Modifier – Meaning**

**Observe the following code.**

*class Test{*

*public void display() {   
}*

*}*

*class Demo extends Test*

*{*

*public void display() {*

*}*

*}*

In the above code, the **display()** method of **Test** class is overridden by **Demo** class. In fact, **Demo** class is at liberty to override or not. Now let us apply a small modifier to **display()** method of **Test** class. See the following code.

***class Test***

***{***

***public final void display() { }***

***}***

***class Demo extends Test***

***{***

***public void display() { }***

***}***

In the super class **Tes**t, observe, the **display()** method is added with **final** keyword. In the super class, if a method is declared as **final**, it cannot be overridden by subclass. That is, super class by declaring method as **final** does not allow the subclass to override. This is the modification given to the method in super class with **final**. **final** is known as access modifier.

Now let us see one more modifier to have better understanding. Observe the following code.

***class Demo{***

***int marks = 50;***

***static int price = 70;***

***public static void main(String args[]){***

***Demo d1 = new Demo();***

***System.out.println("d1.marks: "+ d1.marks);***

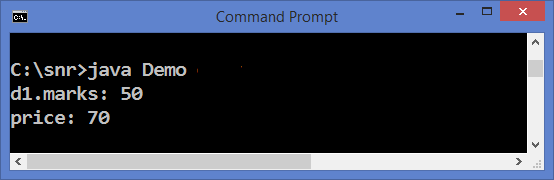
*// marks called with object d1*

***System.out.println("price: " + price);***

*// price called without object d1*

***}***

***}***



In **Demo** class, marks is a **non-static** variable and **price** is static variable. **marks** (non-static variable) requires object to call from main() method where as **static** variable **price** does not require object. So, now what is the modification **static** gave. **static** modifies the access and gives the facility to call without the help of an object.

**Access Specifiers Modifiers Java**

Following list gives the details of access specifiers and modifiers that can be applied to classes, variables and methods.

**Class Specifiers & Modifiers**

Three modifiers exist that can be applied to classes. Modifier comes just before the class name.

|  |  |
| --- | --- |
| **MODIFIER KEYWORD** | **MEANING** |
| [**abstract**](http://way2java.com/oops-concepts/abstract-classes/) | Objects cannot be created |
| [**final**](http://way2java.com/oops-concepts/many-meanings-of-final-keyword-%e2%80%93-in-and-outs/) | cannot be inherited |
| [**strictfp**](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | Guarantees the same precision for floating-point values in the class irrespective of the OS on which the program is executed |

The two [access specifiers](http://way2java.com/packages/access-specifiers-accessibility-permissions-restrictions/) a class can accept are **public**or **default**. **Default means the specifier is not mentioned at all**.

**Variable Specifiers & Modifiers**

Four modifiers and four specifiers exist that can be applied to variables.

|  |  |
| --- | --- |
| **MODIFIER KEYWORD** | **MEANING** |
| [**static**](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/) | Also known as "class variable". No object is necessary to call |
| [**final**](http://way2java.com/oops-concepts/many-meanings-of-final-keyword-%e2%80%93-in-and-outs/) | cannot be reassigned |
| [**transient**](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | Not serialized |
| [**volatile**](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | Value is more liable to change |

All the [four access specifiers](http://way2java.com/oops-concepts/access-specifiers-access-modifiers/give%20strictfp%20link%20in%20Access%20Specifiers%20&%20Access%20Modifiers%20of%20oops), Java supports, can be applied to variables – **public**, **protected**,**default**and **private**. If the specifier is not mentioned, it takes default access. **A local variable must be default only.**

**Method Specifiers & Modifiers**

Six modifiers exist that can be used with methods.

|  |  |
| --- | --- |
| MODIFIER KEYWORD | **MEANING** |
| [final](http://way2java.com/oops-concepts/many-meanings-of-final-keyword-%e2%80%93-in-and-outs/) | Subclass cannot override |
| [abstract](http://way2java.com/oops-concepts/abstract-classes/) | No method body exists |
| [native](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | Java method takes the help of underlying OS |
| [static](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/) | Object is not necessary to call |
| [synchronized](http://way2java.com/multithreading/synchronization-and-deadlock) | Used to lock the source and renders a thread-safe operation |
| [strictfp](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | Guarantees the same precision for floating-point values in the class irrespective of the OS on which the program is executed |

All the four access specifiers, **public**, **protected**, **default**and **private**, can be applied to methods. If not mentioned any access specifier, JVM takes default access.

**Access Modifiers :**

Access modifiers in java are used to control the visibility of a field, method, class and constructor.

There are 4 access modifiers in java. They are :

**1). Private**

**2). Default or Package**

**3). Protected**

**4). Public**

***1) Private***

Private members of class in not accessible anywhere in program these are only accessible within the class.

Private are also called **class level access modifiers.**

**Usage of Private members :**

Private members of a class whether it is a field or method or constructor cannot be accessed outside the class.

**Inheritance of Private Members :**

Private members will not be inherited to sub class.

**Important Note :**

1). Class cannot be a private except inner classes. Inner classes are nothing but again members of outer class. So members of a class (field, method, constructor and inner class) can be private but not the class itself.

2). We can’t create sub classes to that class which has only private constructors.

**Examples:**

***class*** *Hello{*

***private******int*** *a=20;*

***private******void*** *show(){*

*System.****out****.println("Hello java");*

*}*

*}*

***public******class*** *Demo{*

***public******static******void*** *main(String args[]) {*

*Hello obj=****new*** *Hello();*

*System.****out****.println(obj.a);*

***//Compile Time Error, you can't access private data***

*obj.show();*

***//Compile Time Error, you can't access private methods***

*}*

*}*

*class A{*

*private int i;*

*private void methodOfClassA(){* ***//Private Method***

*System.out.println(i);* ***//Private field can be used within class***

*B b = new B();* ***//Private inner class can be used within class***

*}*

*private class B    {*

***//Private Inner Class***

*}*

*}*

*class C extends A*

*{*

*void methodOfClassC()*

*{*

*//System.out.println(i);* ***Private member cannot be inherited***

*A a = new A();*

*//System.out.println(a.i);* ***Private field cannot be used outside the class***

*//a.methodOfClassA();* ***Private method cannot be used outside the class***

*//A.B b = new A.B();* ***Private inner class cannot be used outside the class***

*}*

*}*

private class A

{

**//Outer class Can not be private**

}

class A{

    private A()    {

**//Private Constructor**

    }

    private A(int i){

**//Private constructor**

    }

}

class B extends A{

**//Can't create subclass to the class**

**//which has only private constructors**

}

### Role of Private Constructor

|  |
| --- |
| If you make any class constructor private, you cannot create the instance of that class from outside the class. For example: |

class A{

private A(){

}//private constructor

void msg(){

System.out.println("Hello java");

}

}

public class Simple{

 public static void main(String args[]){

   A obj=new A**();//Compile Time Error**

 }

}

### 2). Default or Package or No-Access Modifiers

Default members of class is accessible only within same class and other class of same package.

Default are also called package level access modifiers.

**Usage of Default members :**

Default members or members with No-Access modifiers are accessed or visible within the package only. It applies to outer classes also.

**Inheritance Of Default Members :**

Default members can be inherited to sub classes within package.

**Examples:**

**//save by A.java**

**package** pack;

**class** A{

**void** show(){

System.**out**.println("Hello Java");

}

}

**//save by B.java**

**package** pack2;

**import** pack1.\*;

**class** B{

**public** **static** **void** main(String args[]) {

A obj = **new** A(); **//Compile Time Error, can't access outside the package**

obj.show(); **//Compile Time Error, can't access outside the package**

}

}

-------------------------------------------------------------------------------------

package pack1;

class A{

    int i;

    A() {

**//Constructor with default modifier**

    }

    void methodOfClassA()    {

**//Method with default access modifier**

        System.out.println(i);

        B b = new B();

    }

    class B

    {

**//Inner Class with default access modifier**

    }

}

class C extends A{

    void methodOfClassC(){

        System.out.println(i);        **//Default field can be inherited within package**

        A a = new A();

        System.out.println(a.i);     **//Default field can be used within the package**

        a.methodOfClassA();          **//Default method can be used within the package**

        A.B b = new A.B();           **//Default inner class can be used within the package**

    }

}

package pack2;

**//import pack1.A;      Class A with default access modifier not visible outside the package**

**/\*class D extends A      Default Class cannot have sub class outside the package**

{

    void methodOfClassD()    {

        System.out.println(i);        **Default field cannot be inherited outside package**

        A a = new A();           **Can't use constructor with default access modifier outside the package**

        System.out.println(a.i);     **Default field cannot be used outside the package**

        a.methodOfClassA();          **Default method cannot be used outside the package**

        A.B b = new A.B();           **Default inner class cannot be used outside the package**

    }

}\*/

### 3). Protected :

Protected members of class is accessible within the same class and other class of same package and also accessible in inherited class of other package.

Protected are also called **derived level access modifiers**.

**Usage of Protected Member :**

Protected member can be used within the package only.

**Inheritance Of Protected Member :**

Protected Member can be inherited to any sub classes.

**Examples:**

**// save A.java**

**package** pack1;

**public** **class** A{

**protected** **void** show(){

System.**out**.println("Hello Java");

}

}

**//save B.java**

**package** pack2;

**import** pack1.\*;

**class** B **extends** A{

**public** **static** **void** main(String args[]){

B obj = **new** B();

obj.show();

}

}

---------------------------------------------------------------------------------------------------------------------

package pack1;

public class A{

    protected int i;

    protected void methodOfClassA()    {

**//Protected method**

        System.out.println(i); **//Protected field can be used within class**

        B b = new B();         **//Protected Inner Class can be used within class.**

    }

    protected class B

    {

**//Protected Inner Class**

    }

}

class C extends A{

    void methodOfClassC()    {

        System.out.println(i);        **//Protected field can be inherited to any sub class**

         A a = new A();

        System.out.println(a.i);     **//Protected field can be used within the package**

        a.methodOfClassA();          **//Protected method can be used within the package**

        A.B b = new A.B();           **//Protected Inner Class can be used within the package**

    }

}

package pack2;

import pack1.A;

class D extends A

{

    void methodOfClassD()    {

        System.out.println(i);        **//Protected field can be inherited to any sub class**

        A a = new A();

        //System.out.println(a.i);     **Protected field cannot be used outside the package**

        //a.methodOfClassA();          **Protected method cannot be used outside the package**

        //A.B b = new A.B();           **Protected inner class cannot be used outside the package**

    }

}

**Important Note :**

1). Outer class cannot be protected.  
2). We can create sub classes to a class which has only protected constructors but we can’t create objects to that class outside the package.

### 4). Public

**Usage of Public members :**

Public members can be used anywhere.

**Inheritance Of Public Members :**

Public members can be inherited to any sub class.

**class** Hello{

**public** **int** a=20;

**public** **void** show(){

System.**out**.println("Hello java");

}

}

**public** **class** Demo{

**public** **static** **void** main(String args[]) {

Hello obj=**new** Hello();

System.**out**.println(obj.a);

obj.show();

}

}

package pack1;

public class A{

    public int i;

     public void methodOfClassA(){

**//public method**

        System.out.println(i); **//public field can be used anywhere**

        B b = new B();         **//public Inner Class can be used anywhere.**

    }

    public class B{

**//public Inner Class**

    }

}

class C extends A{

    void methodOfClassC()    {

        System.out.println(i);        **//public field can be inherited to any sub class**

         A a = new A();

        System.out.println(a.i);     **//public field can be used anywhere**

        a.methodOfClassA();          **//public method can be used anywhere**

        A.B b = new A.B();           **//public Inner Class can be used anywhere.**

    }

}

package pack2;

import pack1.A;

class D extends A{

    void methodOfClassD()    {

        System.out.println(i);        **//public field can be inherited to any sub class**

        A a = new A();

        System.out.println(a.i);     **//Public field can be used anywhere**

        a.methodOfClassA();          **//Public method can be used anywhere**

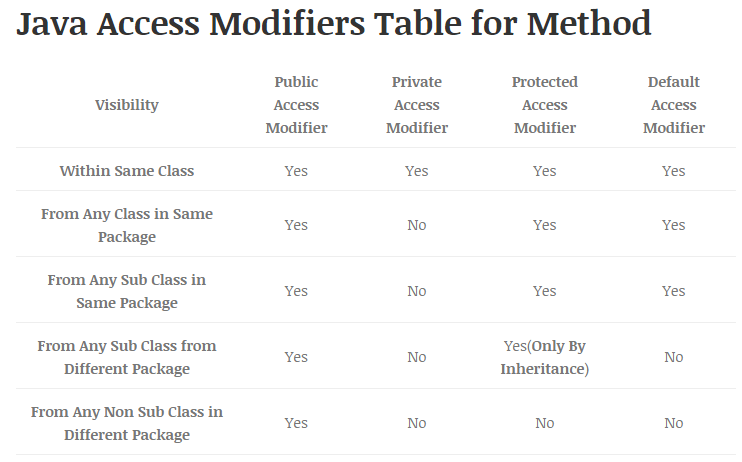
        A.B b = new A.B();           **//Public inner class can be used anywhere**

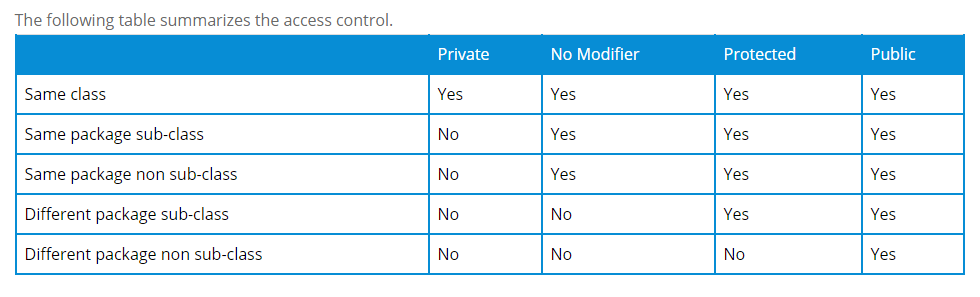
    }

}

**Above concepts can be summarized like below:**

|  |  |  |
| --- | --- | --- |
| **Access Modifier** | **Usage or Access or Visibility** | **Inheritance** |
| **Private** | Within Class Only | Cannot be inherited |
| **Default or No-Access Modifier** | Within Package Only | Can be inherited to sub class within package |
| **Protected** | Within Package Only | Can be inherited to any subclass |
| **Public** | Anywhere | To any subclass |



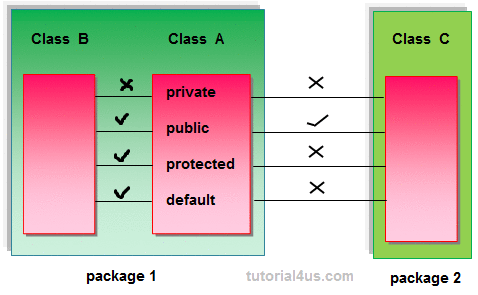


**Note:**

The most strict level of access control allowed in Java is private. Any member variable or method marked as private will be only accessible in that class.

**Note:**

Access modifiers are always used for how to reuse the features within the package and access the package between class to class, interface to interface and interface to class. Access modifiers provides features accessing and controlling mechanism among the classes and interfaces.



**Note:**

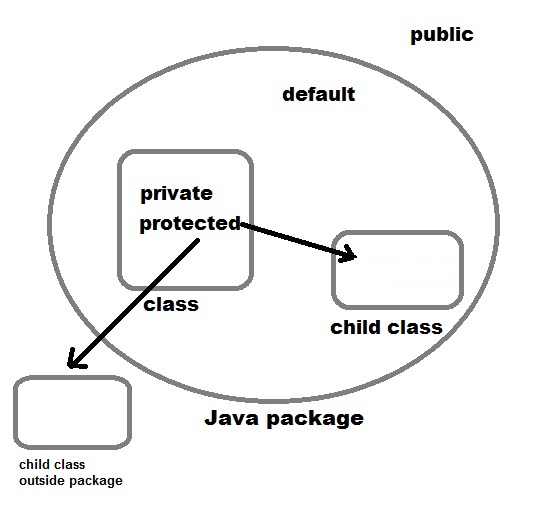
Protected members of class is accessible within the same class and other class of same package and also accessible in inherited class of other package.

**Note:**

private access modifiers is also known as native access modifiers, default access modifiers is also known as package access modifiers, protected access modifiers is also known as inherited access modifiers, public access modifiers is also known as universal access modifiers.

**Note:**

* **Default :** Default has scope only inside the same package
* **Public :** Public scope is visible everywhere
* **Protected :** Protected has scope within the package and all sub classes
* **Private :** Private has scope only within the classes



**Note:**

*The only difference between protected and default is that protected access modifiers respect****class subclass relation****while default does not.*

**Access Control Example included from the book 'Java - The Complete Reference' Eighth Edition, demonstrates the various levels of access.**

**// File : p1\Protection.java**  
package p1;  
class Protection{  
    int n = 1;  
    private int n\_pri = 2;  
    protected int n\_pro = 3;  
    public int n\_pub = 4;  
    public Protection(){  
        System.out.println("base constructor");  
        System.out.println("n = " + n);  
        System.out.println("n\_pri = " + n\_pri);  
        System.out.println("n\_pro = " + n\_pro);  
        System.out.println("n\_pub = " + n\_pub);  
    }  
}

// File : p1\Derived.java  
package p1;  
class Derived extends Protection{  
    public Derived()    {  
        System.out.println("derived constructor");  
        System.out.println("n = " + n);  
        **// System.out.println("n\_pri = " + n\_pri); // WILL NOT WORK**        System.out.println("n\_pro = " + n\_pro);  
        System.out.println("n\_pub = " + n\_pub);  
    }  
}

**// File : p1\SamePackage.java**package p1;  
class SamePackage{  
    public SamePackage()    {  
        Protection p = new Protection();  
        System.out.println("same package constructor");  
        System.out.println("n = " + p.n);  
       **// System.out.println("n\_pri = " + p.n\_pri); // WILL NOT WORK**        System.out.println("n\_pro = " + p.n\_pro);  
        System.out.println("n\_pub = " + p.n\_pub);  
    }  
}

**// File : p2\Protection2.java**  
package p2;  
class Protection2 extends p1.Protection{  
    public Protection2()    {  
        System.out.println("derived other package constructor");  
        **// System.out.println("n = " + n); // WILL NOT WORK  
        // System.out.println("n\_pri = " + n\_pri); // WILL NOT WORK**        System.out.println("n\_pro = " + n\_pro);  
        System.out.println("n\_pub = " + n\_pub);  
    }  
}

**// File : p2\OtherPackage.java**  
package p2;  
class OtherPackage{  
    public OtherPackage()    {  
        p1.Protection p = new p1.Protection();  
        System.out.println("other package constructor");  
**// System.out.println("n = " + n); // WILL NOT WORK  
        // System.out.println("n\_pri = " + n\_pri); // WILL NOT WORK  
        // System.out.println("n\_pro = " + n\_pro); // WILL NOT WORK**        System.out.println("n\_pub = " + n\_pub);  
    }  
}

If you want try this program, first create the files Protection.java, Derived.java and SamePackage.java in the directory p1and Protection2.java and OtherPackage.java in the directory p2.

Then copy the respective code into those java files.   
Create the following Demo.java file to test the classes in package p1.

**// File : p1\Demo.java**  
package p1;  
class Demo{  
    public static void main(String s[])    {  
        Protection ob1 = new Protection();  
        Derived ob2 = new Derived();  
        SamePackage ob3 = new SamePackage();  
    }  
}  
Create the following Demo.java file to test the classes in package p2.

**// File : p2\Demo.java**  
package p2;  
class Demo{  
    public static void main(String s[])    {  
        Protection2 ob1 = new Protection2();  
        OtherPackage ob2 = new OtherPackage();  
    }  
}

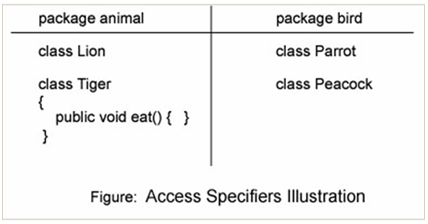
***Access Specifiers Permissions and Restrictions***

**Access specifiers**, as the name indicates, specify the access to our code for other classes – whether other classes can access or not and if permitted, to what extent they can access. Java includes access modifiers also which are quiet different from access specifiers.

Java comes with four access specifiers. They are

1. **public**
2. **protected**
3. **default**
4. **private**

All the Access specifiers are keywords and thereby should be written in lowercase letters only. Java does not have the confusing combinations of public protected etc. which C++ does. All specifiers are straight to their meaning. To understand the specifiers easily, let us take a small scenario where two packages exist – **animal** and **bird**. Let us imagine two classes **Lion** and**Tiger** exist in **animal** package and **Parrot** and **Peacock** exist in **bird** package. It is illustrated in the following figure.

[](http://way2java.com/wp-content/uploads/2011/03/ss17.bmp)

Let us think that there exists a method called **void eat()** in the **Tiger** class of **animal** package. Now the question is who can access this **eat()** method; Lion of same package or Peacock of different package. The access specifiers, in Java, act on **package boundaries**.

1. **public**

Let us assume that the **eat()** method is public. If a member of a class is **public**

1. It can be accessed by the classes of the same package. That is, Lion of the same package can access it.  
2. It can also be accessed by the classes of other packages also. That is, Peacock of other package also can access.

**public** member of a class can be accessed by any class of any package as everyone has got equal right to access. **public** gives maximum permissions among all access Specifiers with no restrictions. **public** means, think like a public park; anybody can come in and anybody can go out; no ticket and no restrictions. Do not think always public, we can put restrictions with other access specifiers. Let us go to the other.

1. **protected**

Let us assume that the **eat()** method is protected. If a member of a class is **protected**

1. It can be accessed by the classes of the same package as if public. That is, Lion of the same package can access it.  
2. Only the subclasses of other packages can access. That is, Peacock should extend Tiger class and use eat() method, else not permitted.

**protected** gives a small restriction for the classes of other package only, but not to the classes of the same package.

1. **default**

Let us assume that the **eat()** method is default. **Default means we do not specify any specifier at all**; that is we write simply **void eat()** without any specifier.

Generally, novices think default is public; it is not correct. Default has its own restrictions.

If a member of a class is **default.**

1. It can be accessed by the classes of the same package as if public. That is, Lion of the same package can access it.  
2. It is impossible for the classes of other package to access (even if they extend also). That is, Peacock cannot access.

The specifiers, public, protected and default does not make any difference for the classes of the same package, but makes to the classes of other packages only. We can say that access specifiers act on package boundaries.

The **default** specifier is known as **package-level access** specifier as the classes of the same package only are allowed to access.

**Note:** Actually there exist a default keyword in Java, but used with **switch** statement. Do not get confused here with the default specifier.

1. **private**

Let us assume that the **eat()** method is private in Tiger class.

If a member of a class is **private**, it can be accessed by the objects of the same class only but not by the classes of the other package or even of the same package. **private** means, it is purely for the private usage of the same class. Other classes, even of the same package, cannot access by composition. That is, private member should be used within the same class with the same class objects.

***Rules of Access Specifiers in Method Overriding***

###### **Access Specifiers Method Overriding: The only rule says:**

**"The subclass overridden method cannot have weaker access than super class method".**

###### Let us see the above rule practically in Access Specifiers Method Overriding.

*class Test{*

*protected void display(){* ***// protected specifier***

*System.out.println("Hello 1");*

*}*

*}*

*public class Demo extends Test{*

*void display(){* ***// overridden with default specifier***

*System.out.println("Hello 2");*

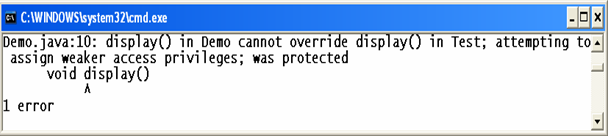
*}*

*public static void main(String args[]) {*

*new Demo().display();*

*}*

*}*



**Protected display()** method of **Test** class is [overridden](http://way2java.com/oops-concepts/method-overriding/) by **Demo** class with **default (not specified at all)** access. It raises compilation error. The subclass can have either same or stronger access like public or same protected. But, it cannot have default or private.

**Why the methods of an interface should be overridden with public only?**

With the above rule, now you get the answer. The methods of an interface must be public, if not mentioned, takes by default as public. So, the subclass which implements the interface should override with the same specifier or more but not less. No stronger specifier than public exists in Java. For this, reason, all the methods of interface must be overridden with public only.

**Access Modifiers and Non-Access Modifiers:**

Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

* [***Java Access Modifiers***](http://www.tutorialspoint.com/java/java_access_modifiers.htm)
* [***Non Access Modifiers***](http://www.tutorialspoint.com/java/java_nonaccess_modifiers.htm)

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones) −

*public* class className {

// ...

}

*private* boolean myFlag;

*static final* double weeks = 9.5;

*protected static final* int BOXWIDTH = 42;

*public static* void main(String[] arguments) {

// body of method

}

## Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

* Visible to the package, the default. No modifiers are needed.
* Visible to the class only (private).
* Visible to the world (public).
* Visible to the package and all subclasses (protected).

## Non Access Modifiers:

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.

**Java Access Modifiers :**

**Default Access Modifier - No keyword:**

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

**Example:**

Variables and methods can be declared without any modifiers, as in the following examples:

*String version = "1.5.1";*

*boolean processOrder() {*

*return true;*

*}*

**Private Access Modifier - private:**

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

**Example:**

The following class uses private access control:

*public class Logger {*

*private String format;*

*public String getFormat() {*

*return this.format;*

*}*

*public void setFormat(String format) {*

*this.format = format;*

*}*

*}*

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and*setFormat(String)*, which sets its value.

***Public Access Modifier - public:***

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However if the public class we are trying to access is in a different package, then the public class still need to be imported.

Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

**Example:**

The following function uses public access control:

*public static void main(String[] arguments) {*

*// ...*

*}*

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

**Protected Access Modifier - protected:**

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

**Example:**

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method:

*class AudioPlayer {*

*protected boolean openSpeaker(Speaker sp) {*

*// implementation details*

*}*

*}*

*class StreamingAudioPlayer {*

*boolean openSpeaker(Speaker sp) {*

*// implementation details*

*}*

*}*

Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used *protected* modifier.

**Access Control and Inheritance:**

The following rules for inherited methods are enforced:

* Methods declared public in a superclass also must be public in all subclasses.
* Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
* Methods declared private are not inherited at all, so there is no rule for them.

# Java Non-Access Modifiers

Below are the Non Access Modifiers available in Java.

* **Final**
* **Abstract**
* **Static**
* **Strictfp**
* **Native**
* **Synchronized**
* **Transient**

***The static Modifier:***

**Static Variables:**

The *static* key word 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 Methods:**

The static key word 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.

**Example:**

The static modifier is used to create class methods and variables, as in the following example:

*public class InstanceCounter {*

*private static int numInstances = 0;*

*protected static int getCount() {*

*return numInstances;*

*}*

*private static void addInstance() {*

*numInstances++;*

*}*

*InstanceCounter() {*

*InstanceCounter.addInstance();*

*}*

*public static void main(String[] arguments) {*

*System.out.println("Starting with " +*

*InstanceCounter.getCount() + " instances");*

*for (int i = 0; i < 500; ++i){*

*new InstanceCounter();*

*}*

*System.out.println("Created " +*

*InstanceCounter.getCount() + " instances");*

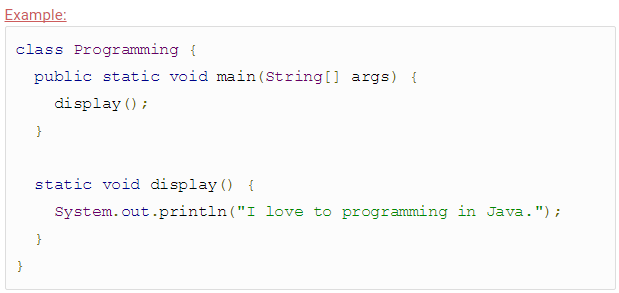
*}*

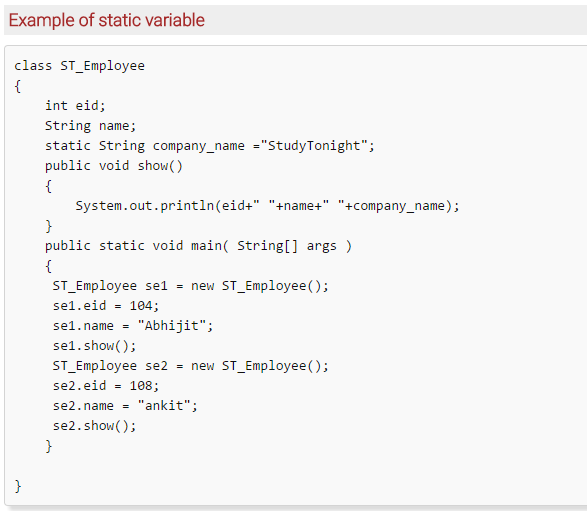
*}*

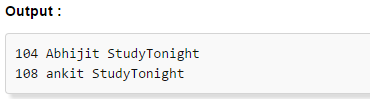
This would produce the following result:

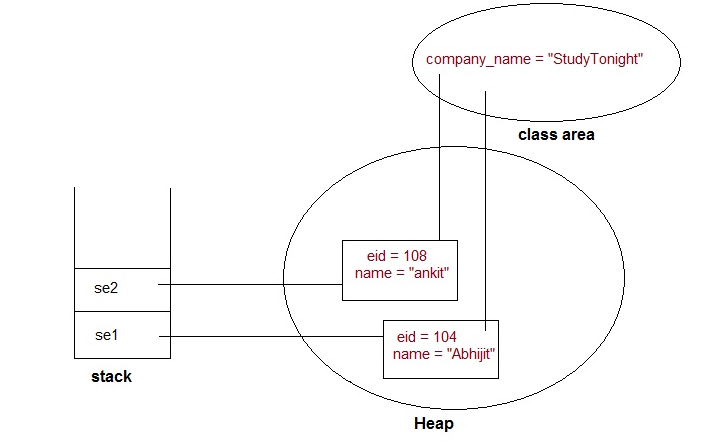
Started with 0 instances

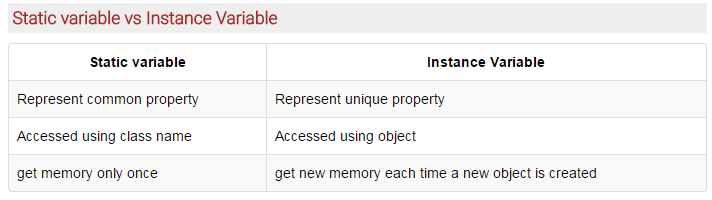
Created 500 instances

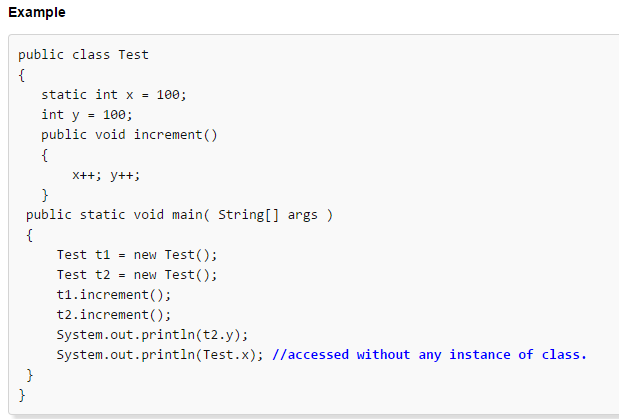
******

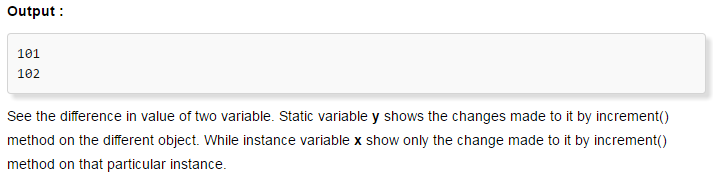
******

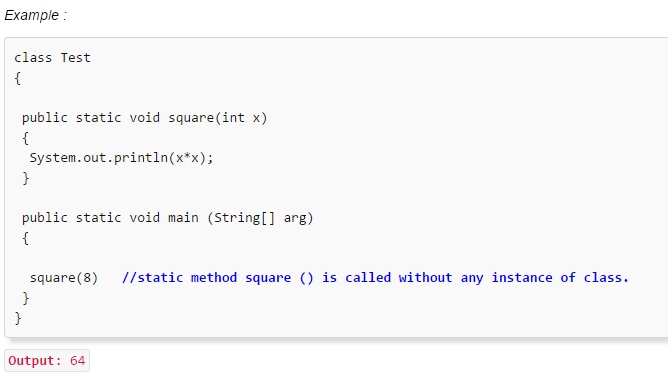
******

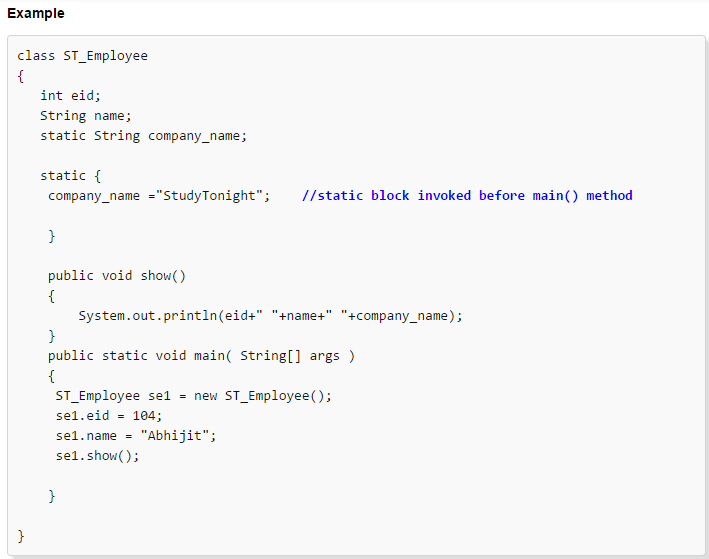
******

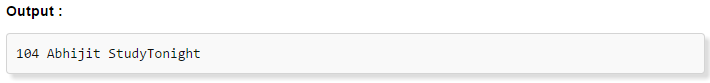
******

******

******

******

******

******

***The final Modifier:***

### Final modifiers are applicable to :

1. Class
2. Method
3. Instance Variable
4. Local Variable
5. Method arguments

### final Variables:

*When a variable is set to final, its value*cannot*be****changed****. Final variables are like constants.*

**Example *:***

*public static final int i = 10;*

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.

**Example:**

***public 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 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.

**Example:**

You declare methods using the *final* modifier in the class declaration, as in the following example:

*public class Test{*

*public final void changeName(){*

*// body of method*

*}*

*}*

**final Classes:**

*A Class when set to final*cannot*be****extended****by any other Class.*

#### ****Example:****

#### ****A String**** Class in java.lang package

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.

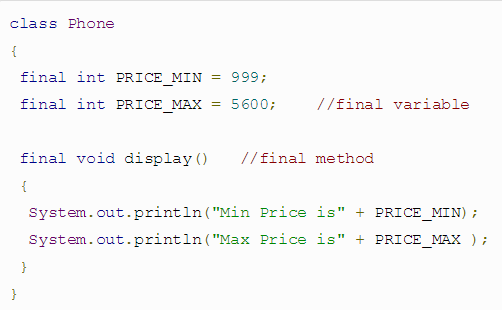
**Example:**

*public final class Test {*

*// body of class*

*}*

**Example:**

****

**The abstract Modifier:**

### Abstract modifiers are applicable to:

1. **Class**
2. **Method**

**abstract Class:**

An abstract Class can have abstract Methods. A Class can also be an abstract class without having any abstract Methods in it. If a Class has an abstract Method , the Class becomes an abstract 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.

**Example:**

***abstract class Caravan{***

***private double price;***

***private String model;***

***private String year;***

***public abstract void goFast(); //an abstract method***

***public abstract void changeColor();***

***}***

**abstract Methods:**

Abstract Methods are those Methods which does not have a body but only a signature.

**Example :**

public abstract void method();

An abstract method is a method declared without 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();

**Example:**

*public abstract class SuperClass{*

*abstract void m(); //abstract method*

*}*

*class SubClass extends SuperClass{*

*// implements the abstract method*

*void m(){*

*.........*

*}*

*}*

**The synchronized Modifier:**

### Synchronized modifiers are applicable to

1. **Method**

*“Synchronized Methods can be accessed by only one thread at a time.”*

The synchronized key word 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.

**Example:**

*public synchronized void showDetails(){*

*.......*

*}*

## Native Non Access Modifier

### Native modifiers are applicable to

1. Method

### Native Method

*Naive Methods indicate that the method is implemented on a platform dependent code.*

## Strictfp Non Access Modifier

### Strictfp modifiers are applicable to

1. Class
2. Method

### Strictfp Class / Method

*Strictfp non access modifier forces floating point or floating point operation to adhere to IEEE 754 standard.*

***Note\*: Strictfp****non access modifier*cannot*be applied on a variable.*

**The transient Modifier:**

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

**Example:**

***public transient int limit = 55; // will not persist***

***public int b; // will persist***

**The volatile Modifier:**

The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

**Example:**

*public class MyRunnable implements Runnable{*

*private volatile boolean active;*

*public void run(){*

*active = true;*

*while (active){ // line 1*

*// some code here*

*}*

*}*

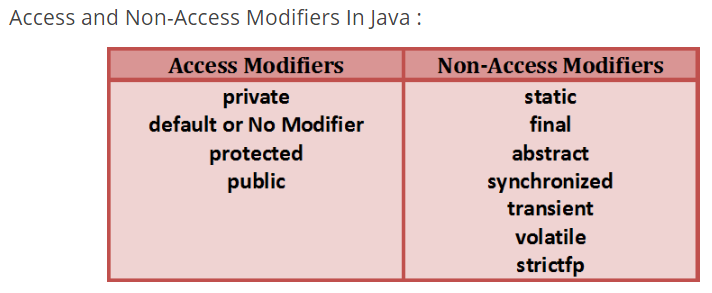
*public void stop(){*

*active = false; // line 2*

*}*

*}*

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1 the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

****

**Access Modifiers Java Meaning Functionality**

In general, an access modifier works between the classes of the same application or within the classes of same package.

**Note:**I have seen some people calling access modifiers also as specifiers. It seems for them Java does not have access modifiers.

To understand better Access Modifiers Java, let us take the examples of [static](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/) and [final](http://way2java.com/oops-concepts/many-meanings-of-final-keyword-%e2%80%93-in-and-outs/) access modifiers. Generally a method or variable is called with an object, but declaring static, you do not require the object. Similarly is with final modifier also. Declaring a method final, the subclass is not allowed to override the super method and a final variable cannot be reassigned. [Synchronized](http://way2java.com/multithreading/synchronization-and-deadlock/) gives a meaning of multiple threads cannot access the same method at a time. Now, you understand the meaning of "modifying the access".

The access modifiers supported by Java are [static](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/), [final](http://way2java.com/oops-concepts/many-meanings-of-final-keyword-%e2%80%93-in-and-outs/), [abstract](http://way2java.com/oops-concepts/abstract-classes/), [synchronized](http://way2java.com/multithreading/synchronization-and-deadlock/), native, volatile, transient and strictfp. Many are discussed at appropriate places in very detail and a few are explained hereunder.

###### **1. native**

"**native**" keyword applies to methods. The code of a native method is written in some other language like C or C++. At execution time, this native code is executed separately and put with the Java output and is given. To do all this, Java includes Java Native Interface (JNI) which is inbuilt into JDK. “native” permits the Java developer to write the machine-dependent code in other language (like C/C++) and make use in Java. Alternatively, if a C or C++ function exists with a very complex code, it is not required to convert it into Java code and instead can be used directly in a Java program.

**Observe some methods of Java API.**

***public static native void sleep(long) throws InterruptedException;***

***// belonging to Thread class***

***public native int read() throws IOException;***

***// belonging to FileInputStream class***

Observe **native keyword**in the above two statements. The functions' code is written in other language (means, not in Java). To execute these methods, Java takes the help of underlying OS.

###### **2. volatile**

The keyword **volatile** applies to variables and objects. A volatile variable value is more likely to get changed in the code. A volatile variable is treated differently by the JVM.

Volatile variables are not optimized (to minimize execution time) for **performance** by the compiler as their values are expected to get changed at any time without information. Volatile is better used with multiple threads that can change a variable value often.

Volatile can be used as follows.

***public volatile int rate = 10;***

Volatile does not have any meaning when applied to final variables or immutable objects or synchronized block.

**The following two statements give compilation error.**

***final volatile int x = 10;  
volatile final int x = 10;***

###### **3. transient Modifier**

It is used with [Serialization](http://way2java.com/serialization/what-is-serialization/).

"**transient**" is a keyword which means the data is not serialized. In object serialization, the objects along with the data are serialized. If the data is not required to be serialized and thereby not written to a file (or sent to the remote client/server), then declare the data as**transient**.

**Following is the way how to declare a variable as transient.**

***public transient double goldRate = 210.5;***

A program on transient keyword is available at [Java Serialization Example](http://way2java.com/serialization/java-serialization-example/).

###### **4. strictfp**

A **floating-point value**, in a language, is platform-dependent. That is, the same floating value, in a Java program, executed on different operating systems may give different outputs (precision). To get the same precision (regardless of hardware, software and OS) on every operating system, declare the class or method as strictfp. "strictfp" is a keyword added in JDK 1.2 version.

"strictfp" is abbreviation for "strict floating-point".

*public strictfp class Demo*

*public strictfp interface Demo*

*public strictfp void display()*

If a class is strictfp, all the code in the class is evaluated with the strict floating point precision as laid in IEEE 754 standards. strictfp follows the rules formatted by IEEE 754. JVM does not apply its own precision.

**IEEE and IEEE 754**

IEEE is an abbreviation for Institute of Electrical and Electronics Engineers. It is a service-oriented body managed by professional engineers. It is started in 1963 and operated in 150 countries with over 3.5 lakh members. It is an amalgamated body of Institute of Radio Engineers and American Institute of Electrical Engineers. It sets standards for many like for floating-point values used by CPU.

**Following table gives the list of access specifiers and modifiers Java that can be applied to variables, methods and classes.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SPECIFIER/MODIFIER** | **LOCAL VARIABLE** | **INSTANCE VARIABLE** | **METHOD** | **CLASS** |
| **Public** | NA | A | A | A |
| **protected** | NA | A | A | NA |
| **default** | A | A | A | A |
| **private** | NA | A | A | NA |
| **Final** | A | A | A | A |
| **Static** | NA | A | A | NA |
| **synchronized** | NA | NA | A | NA |
| **Native** | NA | NA | A | NA |
| **volatile** | NA | A | NA | NA |
| **transient** | NA | A | NA | NA |
| **strictfp** | NA | NA | A | A |

A: Allowed NA: Not Allowed

**SCJP:**

***Class Modifiers***

Whenever we are writing our own classes compulsory we have to provide some information about our class to the jvm. Like

**1. Whether this class can be accessible from anywhere or not.**

**2. Whether child class creation is possible or not.**

**3. Whether object creation is possible or not etc.**

We can specify this information by using the corresponding modifiers.

The only applicable modifiers for Top Level classes are:

**1. Public**

**2. Default**

**3. Final**

**4. Abstract**

**5. Strictfp**

If we are using any other modifier we will get compile time error.

**Example:**

***private class Test{***

***public static void main(String args[]){***

***int i=0;***

***for(int j=0;j<3;j++){***

***i=i+j;***

***}***

***System.out.println(i);***

***}***

***}***

**OUTPUT:**

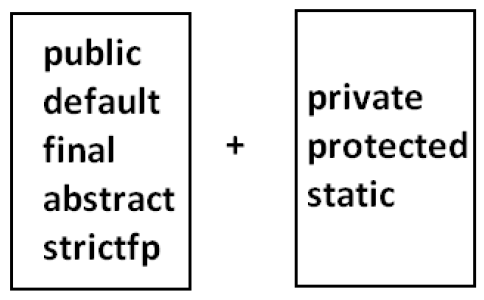
***Compile time error.***

***D:\Java>javac Test.java***

***Test.java:1: modifier private not allowed here private class Test***

But For the inner classes the following modifiers are allowed.

***Diagram:***

****What is the difference between access specifier and access modifier ?**

* In old languages 'C' (or) 'C++' public, private, protected, default are considered as access specifiers and all the remaining are considered as access modifiers.
* But in java there is no such type of division all are considered as access

modifiers.

**Public Classes:**

If a class declared as public then we can access that class from anywhere. Within the package or outside the package.

**Example:**

**Program1:**

package pack1;

public class Test{

public void methodOne(){

System.out.println("test class methodone is executed");

}

}

**Compile the above Program:**

D:\Java>javac -d . Test.java

**Program2:**

*package pack2;*

*import pack1.Test;*

*class Test1{*

*public static void main(String args[]){*

*Test t=new Test();*

*t.methodOne();*

*}*

*}*

**OUTPUT:**

*D:\Java>javac -d . Test.java*

*D:\Java>javac -d . Test1.java*

*D:\Java>java pack2.Test1*

*Test class methodone is executed.*

If class Test is not public then while compiling Test1 class we will get compile time error saying pack1.Test is not public in pack1; cannot be accessed from outside package.

**Default Classes:**

If a class declared as the default then we can access that class only within the current package hence default access is also known as "package level access".

**Example:**

**Program 1:**

*package pack1;*

*class Test{*

*public void methodOne(){*

*System.out.println("test class methodone is executed");*

*}*

*}*

**Program 2:**

*package pack1;*

*import pack1.Test;*

*class Test1{*

*public static void main(String args[]){*

*Test t=new Test();*

*t.methodOne();*

*}*

*}*

**OUTPUT:**

D:\Java>javac -d . Test.java

D:\Java>javac -d . Test1.java

D:\Java>java pack1.Test1

Test class methodone is executed

***Final Modifier:***

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

**Final 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.

**Example:**

**Program 1:**

***class Parent{***

***public void property(){***

***System.out.println("cash+gold+land");***

***}***

***public final void marriage(){***

***System.out.println("subbalakshmi");***

***}***

***}***

**Program 2:**

class child extends Parent{

public void marriage(){

System.out.println("Thamanna");

}

}

**OUTPUT:**

*Compile time error.*

*D:\Java>javac Parent.java*

*D:\Java>javac child.java*

*child.java:3: marriage() in child cannot override marriage() in Parent; overridden method is final public void marriage(){*

***Final Class:***

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

**Example:**

**Program 1:**

**final class Parent**

**{**

**}**

**Program 2:**

**class child extends Parent**

**{**

**}**

**OUTPUT:**

**Compile time error.**

**D:\Java>javac Parent.java**

**D:\Java>javac child.java**

**child.java:1: cannot inherit from final Parent**

**class child extends Parent**

**Note:** Every method present inside a final class is always final by default whether we are declaring or not. But every variable present inside a final class need not be final.

**Example:**

final class parent{

static int x=10;

static{

x=999;

}

}

The main advantage of final keyword is we can achieve security.

**Whereas the main disadvantage is we are missing the key benefits of oops:**

polymorsim (because of final methods), inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyboard.

***Abstract Modifier:***

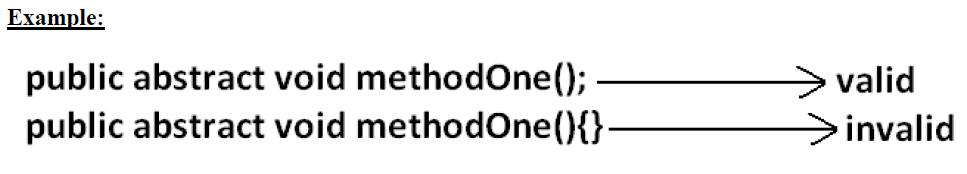
Abstract is the modifier applicable only for methods and classes but not for variables.

**Abstract 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.

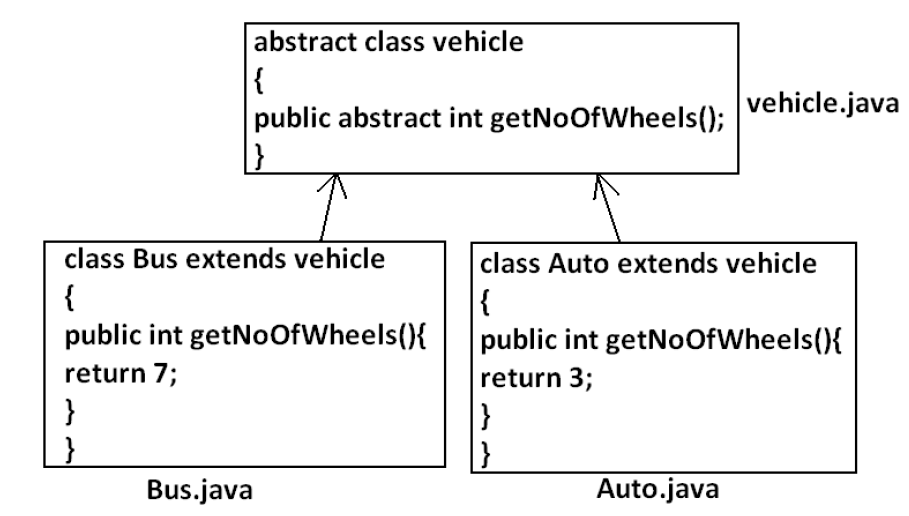
**

Child classes are responsible to provide implementation for parent class abstract

methods.

**Example:**

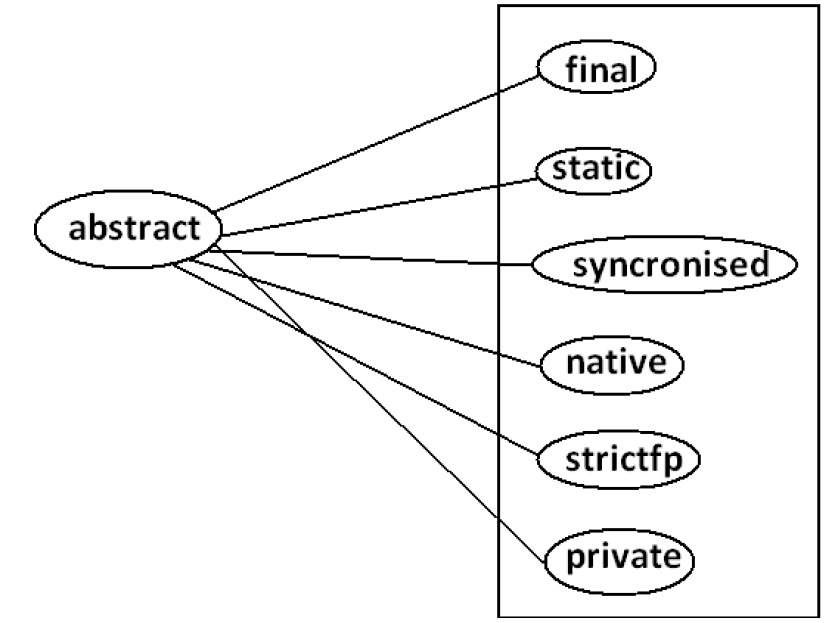
**Program:**

**

* The main advantage of abstract methods is , by declaring abstract method in parent class we can provide guide lines to the child class such that which methods they should compulsory implement.
* Abstract method never talks about implementation whereas if any modifier talks about implementation then the modifier will be enemy to abstract and that is always illegal combination for methods.

**The following are the various illegal combinations for methods.**

**Diagram:**

**

**All the 6 combinations are illegal.**

***Abstract 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.

**Example:**

abstract class Test{

public static void main(String args[]){

Test t=new Test();

}

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:4: Test is abstract; cannot be instantiated

Test t=new Test();

**What is the difference between abstract class and abstract method ?**

* If a class contain at least on abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
* Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.

**Example1:**

HttpServlet class is abstract but it doesn't contain any abstract method.

**Example2:**

Every adapter class is abstract but it doesn't contain any abstract method.

**Example1:**

class Parent

{

public void methodOne();

}

**Output:**

Compile time error.

D:\Java>javac Parent.java

Parent.java:3: missing method body, or declare abstract

public void methodOne();

**Example2:**

class Parent

{

public abstract void methodOne(){}

}

**Output:**

Compile time error.

Parent.java:3: abstract methods cannot have a body

public abstract void methodOne(){}

**Example3:**

class Parent

{

public abstract void methodOne();

}

**Output:**

Compile time error.

D:\Java>javac Parent.java

Parent.java:1: Parent is not abstract and does not

override abstract method methodOne() in Parent

class Parent

If a class extends any abstract class then compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

**Example:**

abstract class Parent

{

public abstract void methodOne();

public abstract void methodTwo();

}

class child extends Parent

{

public void methodOne(){}

}

**Output:**

Compile time error.

D:\Java>javac Parent.java

Parent.java:6: child is not abstract and does not

override abstract method methodTwo() in Parent

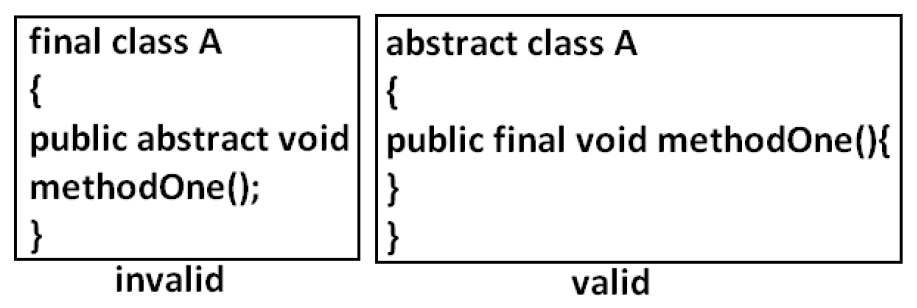
class child extends Parent

If we declare class child as abstract then the code compiles fine but child of child is responsible to provide implementation for methodTwo().

**What is the difference between final and abstract ?**

* For abstract methods compulsory we should override in the child class to provide implementation. Whereas for final methods we can't override hence abstract final combination is illegal for methods.
* For abstract classes we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.
* Final class cannot contain abstract methods whereas abstract class can contain final method.

**Example:**

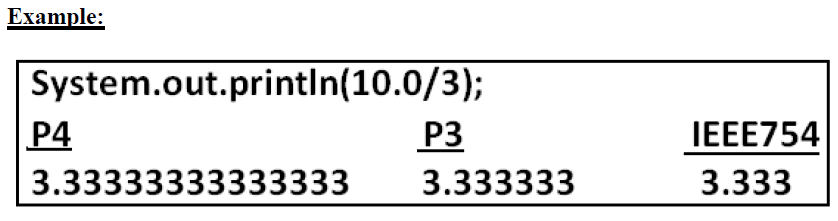
**

**Note:**

Usage of abstract methods, abstract classes and interfaces is always good Programming practice.

**Strictfp:**

* strictfp is the modifier applicable for methods and classes but not for variables.
* Strictfp modifier introduced in 1.2 versions.
* Usually the result of floating point of arithmetic is varing from platform to platform , to overcome this problem we should use strictfp modifier.
* If a method declare as the Strictfp then all the floating point calculations in that method has to follow IEEE754 standard, So that we will get platform independent results.

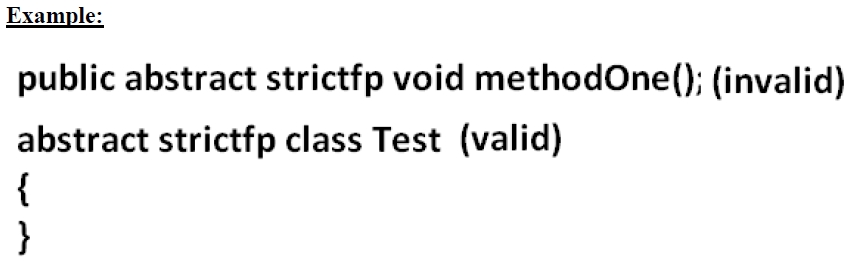
**

If a class declares as the Strictfp then every concrete method(which has body) of that class has to follow IEEE754 standard for floating point arithmetic, so we will get platform independent results.

**What is the difference between abstract and strictfp ?**

* Strictfp method talks about implementation where as abstract method never talks about implementation hence abstract, strictfp combination is illegal for methods.
* But we can declare a class with abstract and strictfp modifier simultaneously.

That is abstract strictfp combination is legal for classes but illegal for methods.

******

**Member modifiers:**

**Public members:**

If a member declared as the public then we can access that member from anywhere "**but the corresponding class must be visible"** hence before checking member visibility we have to check class visibility.

**Example:**

**Program 1:**

package pack1;

class A{

public void methodOne(){

System.out.println("a class method");

}

}

**D:\Java>javac -d . A.java**

**Program 2:**

package pack2;

import pack1.A;

class B{

public static void main(String args[]){

A a=new A();

a.methodOne();

}

}

**Output:**

Compile time error.

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1;

cannot be accessed from outside package import pack1.A;

In the above Program even though methodOne() method is public we can't access from class B because the corresponding class A is not public that is both classes and methods are public then only we can access.

**Default member:**

If a member declared as the default then we can access that member only within the current package hence default member is also known as package level access.

**Example 1:**

**Program 1:**

package pack1;

class A{

void methodOne(){

System.out.println("methodOne is executed");

}

}

**Program 2:**

package pack1;

import pack1.A;

class B{

public static void main(String args[]){

A a=new A();

a.methodOne();

}

}

**Output:**

D:\Java>javac -d . A.java

D:\Java>javac -d . B.java

D:\Java>java pack1.B

methodOne is executed

**Example 2:**

**Program 1:**

package pack1;

class A{

void methodOne(){

System.out.println("methodOne is executed");

}

}

**Program 2:**

package pack2;

import pack1.A;

class B{

public static void main(String args[]){

A a=new A();

a.methodOne();

}

}

**Output:**

Compile time error.

D:\Java>javac -d . A.java

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1; cannot be accessed from outside

Package import pack1.A;

**Private members:**

* If a member declared as the private then we can access that member only within the current class.
* Private methods are not visible in child classes where as abstract methods should be visible in child classes to provide implementation hence private, abstract combination is illegal for methods.r
* ot

**Protected members:**

If a member declared as the protected then we can access that member within the current package anywhere but outside package only in child classes.

***Protected=default+kids.***

We can access protected members within the current package anywhere either by child reference or by parent reference

But from outside package we can access protected members only in child classes and should be by child reference only that is we can't use parent reference to call protected members from outside package.

**Example:**

**Program 1:**

package pack1;

public class A{

protected void methodOne(){

System.out.println("methodOne is executed");

}

}

Program 2:

*package pack1;*

*class B extends A{*

*public static void main(String args[]){*

*A a=new A();*

*a.methodOne();*

*B b=new B();*

*b.methodOne();*

*A a1=new B();*

*a1.methodOne();*

*}*

*}*

**Output:**

*D:\Java>javac -d . A.java*

*D:\Java>javac -d . B.java*

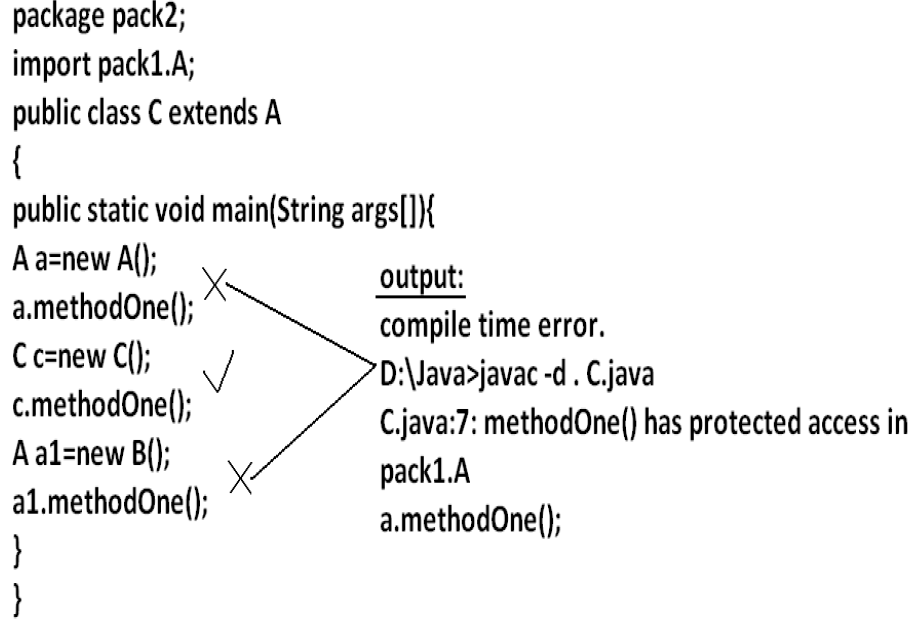
*D:\Java>java pack1.B*

*methodOne is executed*

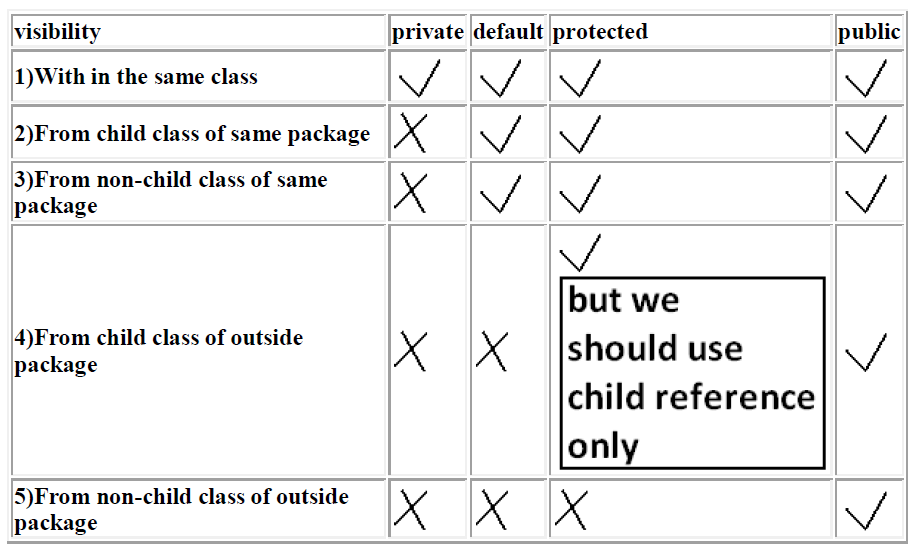
*methodOne is executed*

*methodOne is executed*

**Example 2:**



***Compression of private, default, protected and public:***

**

* **The least accessible modifier is private.**
* **The most accessible modifier is public.**

**Private<default<protected<public**

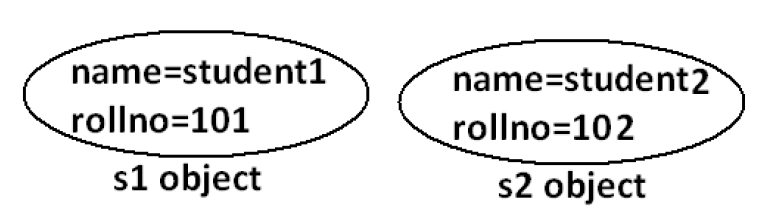
Recommended modifier for variables is private where as recommended modifier for methods is public.

**Final variables:**

**Final instance variables:**

* If the value of a variable is varied from object to object such type of variables are called instance variables.
* For every object a separate copy of instance variables will be created.

**DIAGRAM**:

**

For the instance variables it is not required to perform initialization explicitly jvm will always provide default values.

**Example:**

class Test{

int i;

public static void main(String args[]){

Test t=new Test();

System.out.println(t.i);

}

}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

0

If the instance variable declared as the final compulsory we should perform

initialization explicitly and JVM won't provide any default values.

whether we are using or not otherwise we will get compile time error.

**Example:**

**Program 1:**

class Test

{

int i;

}

**Output:**

D:\Java>javac Test.java

D:\Java>

**Program 2:**

class Test

{

final int i;

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:1: variable i might not have been initialized class Test

**Rule:**

For the final instance variables we should perform initialization before constructor completion. That is the following are various possible places for this.

**1) At the time of declaration:**

**Example:**

class Test

{

final int i=10;

}

**Output:**

D:\Java>javac Test.java

D:\Java>

**2) Inside instance block:**

**Example:**

class Test{

final int i;{

i=10;

}

}

**Output:**

D:\Java>javac Test.java

D:\Java>

**3) Inside constructor:**

**Example:**

class Test{

final int i;

Test(){

i=10;

}

}

**Output:**

D:\Java>javac Test.java

D:\Java>

If we are performing initialization anywhere else we will get compile time error.

**Example:**

class Test{

final int i;

public void methodOne(){

i=10;

}

}

**Output**:

*Compile time error.*

*D:\Java>javac Test.java*

*Test.java:5: cannot assign a value to final variable i,*

*i=10;*

***Final static variables:***

* If the value of a variable is not varied from object to object such type of variables is not recommended to declare as the instance variables. We have to declare those variables at class level by using static modifier.
* In the case of instance variables for every object a seperate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.
* For the static variables it is not required to perform initialization explicitly jvm will always provide default values.

**Example:**

class Test{

static int i;

public static void main(String args[]){

System.out.println("value of i is :"+i);

}

}

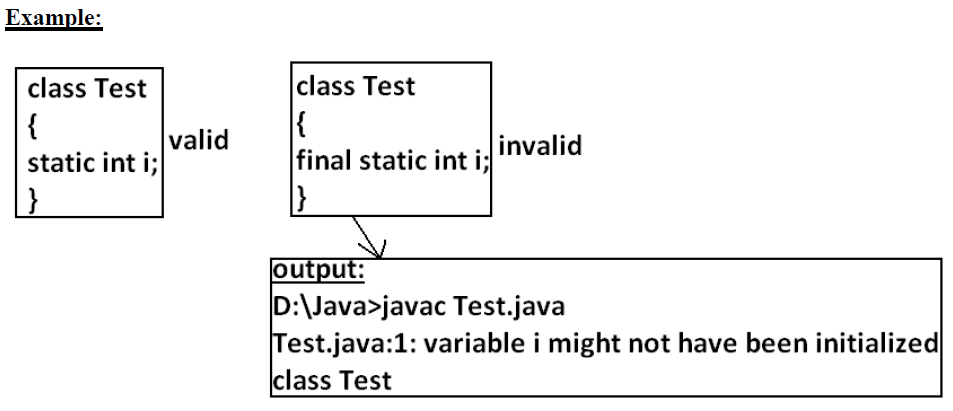
**Output:**

D:\Java>javac Test.java

D:\Java>java Test

Value of i is: 0

If the static variable declare as final then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.(The JVM won't provide any default values).



**Rule:**

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

**1) At the time of declaration:**

**Example:**

class Test{

final static int i=10;

}

**Output**:

D:\Java>javac Test.java

D:\Java>

**2) Inside static block:**

**Example:**

class Test{

final static int i;

static{

i=10;

}

}

**Output:**

Compile successfully.

If we are performing initialization anywhere else we will get compile time error.

**Example**:

class Test{

final static int i;

public static void main(String args[]){

i=10;

}

}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: cannot assign a value to final variable i

i=10;

**Final local variables:**

* To meet temporary requirement of the Programmer sometime we can declare the variable inside a method or block or constructor such type of variables are called local variables.
* For the local variables jvm won't provide any default value compulsory we should perform initialization explicitly before using that variable.

**Example**:

class Test{

public static void main(String args[]){

int i;

System.out.println("hello");

}

}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

Hello

**Example:**

class Test{

public static void main(String args[]){

int i;

System.out.println(i);

}

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:5: variable i might not have been initialized

System.out.println(i);

Even though local variable declared as the final before using only we should perform initialization.

**Example:**

class Test{

public static void main(String args[]){

final int i;

System.out.println("hello");

}

}

**Output:**

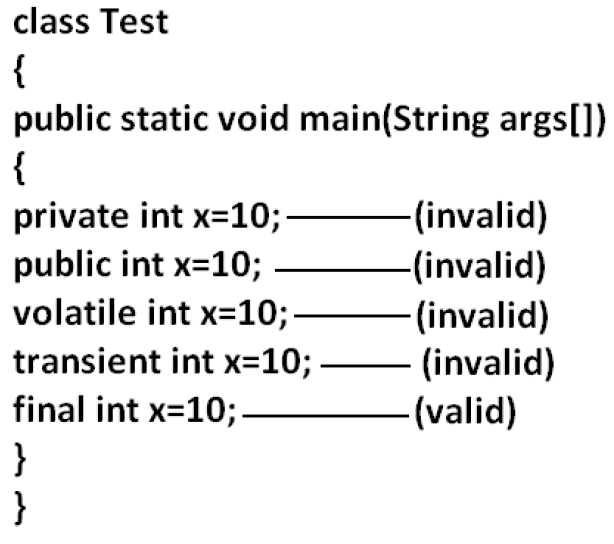
D:\Java>javac Test.java

D:\Java>java Test

Hello

**Note:** The only applicable modifier for local variables is final if we are using any other modifier we will get compile time error.

**Example**:



**Output:**

Compile time error.

D:\Java>javac Test.java

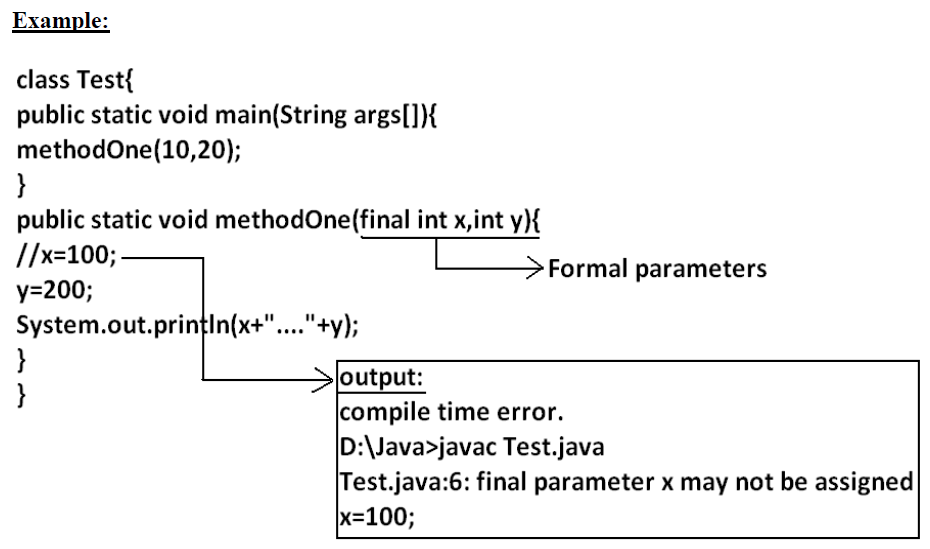
Test.java:5: illegal start of expression

private int x=10;

**Formal parameters:**

* The formal parameters of a method are simply acts as local variables of that method hence it is possible to declare formal parameters as final.
* If we declare formal parameters as final then we can't change its value within the method.

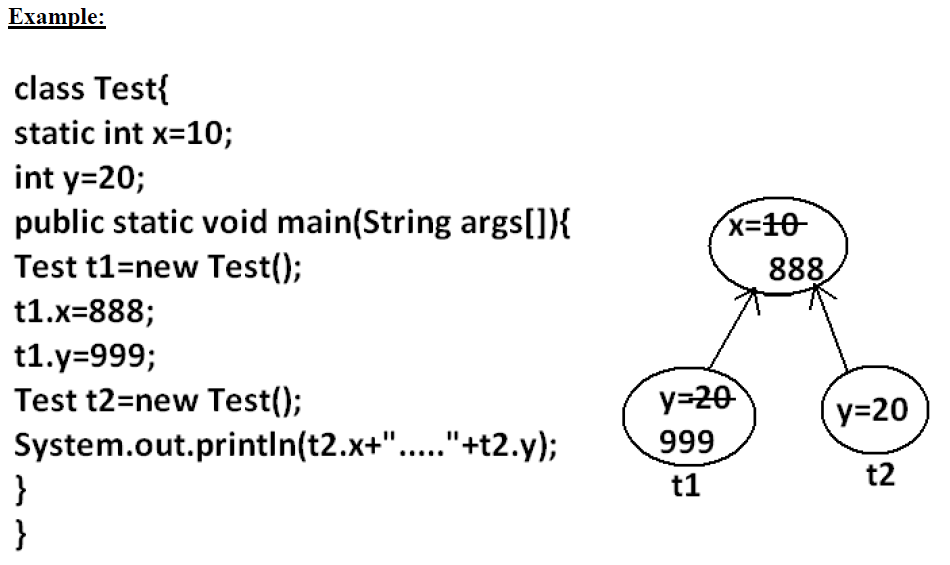
**Example:**



* For instance and static variables JVM will provide default values but if instance and static declared as final JVM won't provide default value compulsory we should perform initialization whether we are using or not .
* For the local variables JVM won't provide any default values we have to perform explicitly before using that variables , this rule is same whether local variable final or not.

**Static modifier:**

* Static is the modifier applicable for methods, variables and blocks.
* We can't declare a class with static but inner classes can be declaring as the static.
* In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by all objects of that class.



**Output:**

D:\Java>javac Test.java

D:\Java>java Test

888.....20

* Instance variables can be accessed only from instance area directly and we can't access from static area directly.
* But static variables can be accessed from both instance and static areas directly.

1. int x=10;
2. Static int x=10;
3. Public void methodOne(){

System.out.println(x);

}

1. Public static void methodOne(){

System.out.println(x);

}

**Which are the following declarations are allow within the same class simultaneously ?**

1. **1 and 3**

**Example:**

***class Test{***

***int x=10;***

***public void methodOne(){***

***System.out.println(x);***

***}***

***}***

**Output:**

Compile successfully.

**b) 1 and 4**

**Example:**

class Test{

int x=10;

public static void methodOne(){

System.out.println(x);

}

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:5: non-static variable x cannot be referenced from a static context System.out.println(x);

**c) 2 and 3**

**Example:**

class Test{

static int x=10;

public void methodOne(){

System.out.println(x);

}

}

**Output**:

Compile successfully.

**d) 2 and 4**

**Example**:

class Test{

static int x=10;

public static void methodOne(){

System.out.println(x);

}

}

**Output:**

Compile successfully.

**e) 1 and 2**

**Example:**

class Test

{

int x=10;

static int x=10;

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:4: x is already defined in Test

static int x=10;

**f) 3 and 4**

**Example:**

class Test{

public void methodOne(){

System.out.println(x);

}

public static void methodOne(){

System.out.println(x);

}

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:5: methodOne() is already defined in Test

public static void methodOne(){

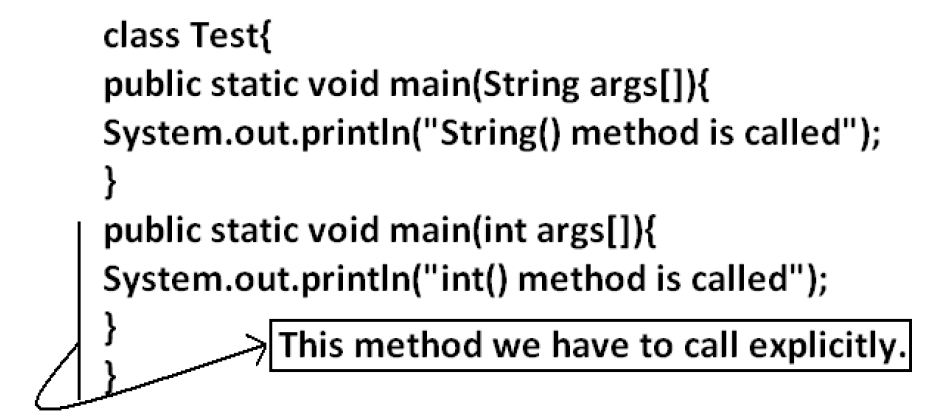
For static methods implementation should be available but for abstract methods implementation is not available hence static abstract combination is illegal for methods.

**Case 1:**

Overloading concept is applicable for static method including main method also.But JVM will always call String[] args main method .

The other overloaded method we have to call explicitly then it will be executed just like a normal method call .

**Example:**



**Output :**

String() method is called

**Case 2:**

Inheritance concept is applicable for static methods including main() method hence while executing child class, if the child doesn't contain main() method then the parent class main method will be executed.

**Example**:

class Parent{

public static void main(String args[]){

System.out.println("parent main() method called");

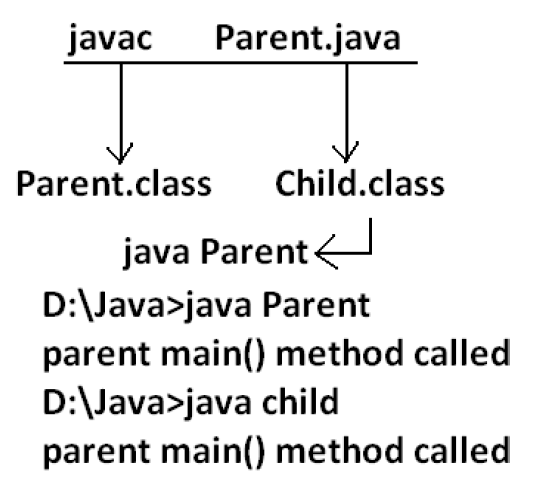
}

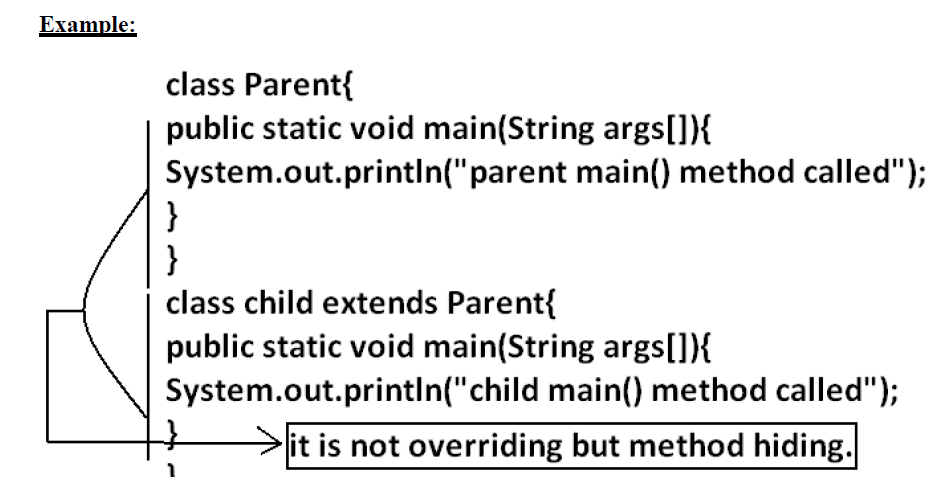
}

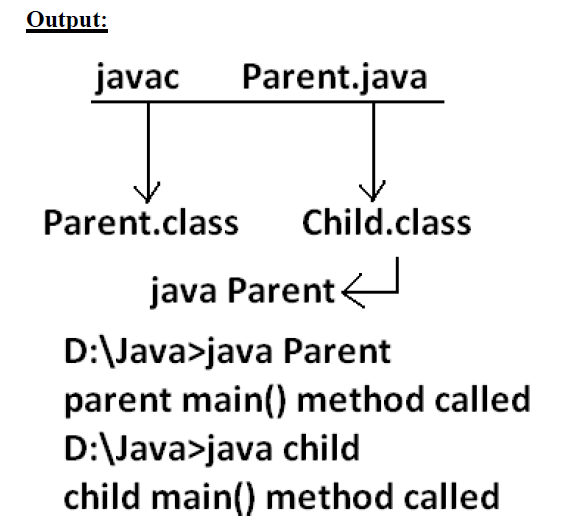
class child extends Parent{

}

**Output:**

****

****

****

* It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

***Native modifier:***

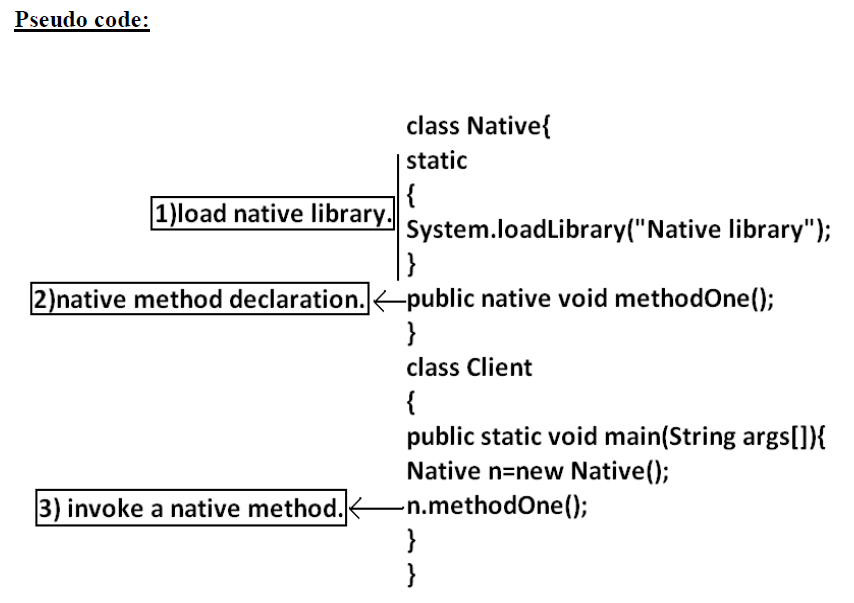
* Native is a modifier applicable only for methods but not for variables and classes.
* The methods which are implemented in non java are called native methods or foreign methods.

**The main objectives of native keyword are:**

* To improve performance of the system.
* To use already existing legacy non-java code.
* To achieve machine level communication(memory level - address)
* Pseudo code to use native keyword in java.

**To use native keyword:**

**Pseudo code:**

****

* For native methods implementation is already available and we are not responsible to provide implementation hence native method declaration should compulsory ends with semicolon.
* *Public native void methodOne()----invalid*
* *Public native void methodOne();---valid*
* For native methods implementation is already available where as for abstract methods implementation should not be available child class is responsible to provide that, hence abstract native combination is illegal for methods.
* We can't declare a native method as strictfp because there is no guaranty whether the old language supports IEEE754 standard or not. That is native strictfp combination is illegal for methods.
* For native methods inheritance, overriding and overloading concepts are applicable.
* The main advantage of native keyword is performence will be improves.
* The main disadvantage of native keyword is usage of native keyword in java breaks platform independent nature of java language.

**Synchronized:**

1. Synchronized is the modifier applicable for methods and blocks but not for variables and classes.
2. If a method or block declared with synchronized keyword then at a time only one thread is allow to execute that method or block on the given object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. But the main disadvantage is it increases waiting time of the threads and effects performance of the system. Hence if there is no specific requirement never recommended to use synchronized keyword.
5. For syncronized methods compulsory implementation should be available , but for abstract methods implementation won't be available , Hence **abstract – synchronized combination** is illegal for methods.

**Transient modifier:**

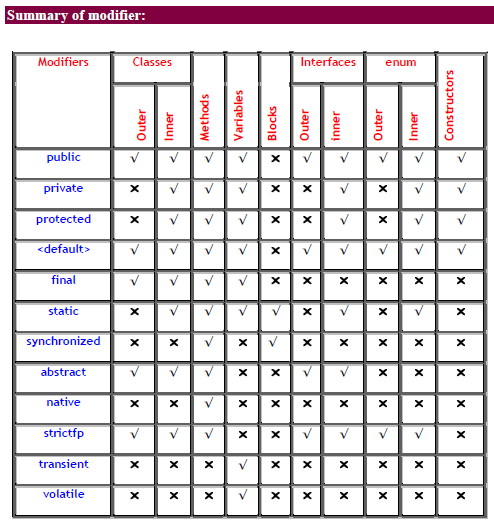
1. Transient is the modifier applicable only for variables but not for methods and classes.
2. At the time of serialization if we don't want to serialize the value of a particular variable to meet the security constraints then we should declare that variable with transient modifier.
3. At the time of serialization jvm ignores the original value of the transient variable and save default value that is transient means "not to serialize".
4. Static variables are not part of object state hence serialization concept is not applicable for static variables duo to this declaring a static variable as transient there is no use.
5. Final variables will be participated into serialization directly by their values due to this declaring a final variable as transient there is no impact.

**Volatile modifier:**

1. Volatile is the modifier applicable only for variables but not for classes and methods.
2. If the value of variable keeps on changing such type of variables we have to declare with volatile modifier.
3. If a variable declared as volatile then for every thread a separate local copy will be created by the jvm, all intermediate modifications performed by the thread will takes place in the local copy instead of master copy.
4. Once the value got finalized before terminating the thread that final value will be updated in master copy.
5. The main advantage of volatile modifier is we can resolve data inconsistency problems, but creating and maintaining a separate copy for every thread increases complexity of the Programming and effects performance of the system.

Hence if there is no specific requirement never recommended to use volatile modifier and it's almost outdated.

1. Volatile means the value keep on changing where as final means the value never changes hence final volatile combination is illegal for variables.



**Conclusions:**

* The Only Applicable Modifiers for Constructors are *public, private, protected,* and *<default>.*
* The Only Applicable Modifiers for Local Variable is *final.*
* The Only Modifier which is applicable for Classes but Not for Interfaces is *final.*
* The Modifiers which are Applicable for Classes but Not for enum are *final* and *abstract.*
* The Modifiers which are Applicable for Inner Classes but Not for Outer Classes are *public, protected,* and *static.*
* The Only Modifier which is Applicable for Methods is *native.*
* The Modifiers which are Applicable for Variables are *transient* and *volatile.*

**Note :**

1. The modifiers which are applicable for inner classes but not for outer classes are

private, protected, static.

2. The modifiers which are applicable only for methods native.

3. The modifiers which are applicable only for variables transient and volatile.

4. The modifiers which are applicable for constructor public, private, protected,

default.

5. The only applicable modifier for local variables is final.

6. The modifiers which are applicable for classes but not for enums are final, abstract.

7. The modifiers which are applicable for classes but not for interface are final.