**static Keyword**

Is used to define class members.

*In a class we can have following 3 types of class members:-*

1. **STATIC DATA MEMBERS**

Represents class attributes. A single copy of static data members is created when the class is loaded. This copy can be shared by all the objects of the class.

**For example -**

**class** A

{

int a, b;

static int c;

-------

-------

}

**When we create objects of this class in some other class what happens is explained in the given adjoining fig1.**

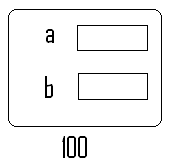
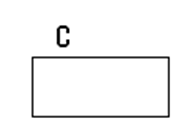
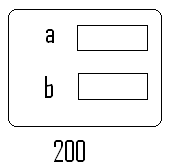
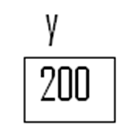
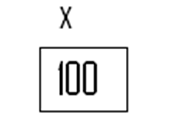


Fig1.

*For the execution of a Java Application, memory is divided into three parts called*

* Stack - LOCAL VARIABLES of methods are stored in STACK.
* Heap - OBJECTS are created in the HEAP.
* Class Area - STATICDATA MEMBERS are saved in CLASSAREA.

1. **STATIC INITIALIZE BLOCK**
2. **STATIC METHODS.**

**class** A

{

**int** a, b;

static int c;

----------

----------

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

{

A x = **new** A();

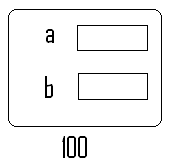
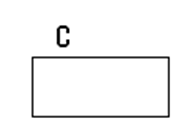
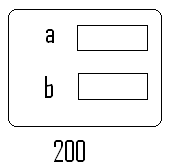
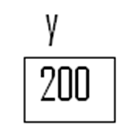
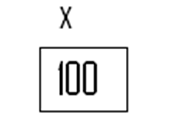
A y = **new** A();

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

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

}

}



Each object of a class contains References of class members

*Memory Representation During the execution of main() method of class A.*

Static Initializer Block- is used to initialize *static data members* of a class.

Syntax –

static

{

Statements

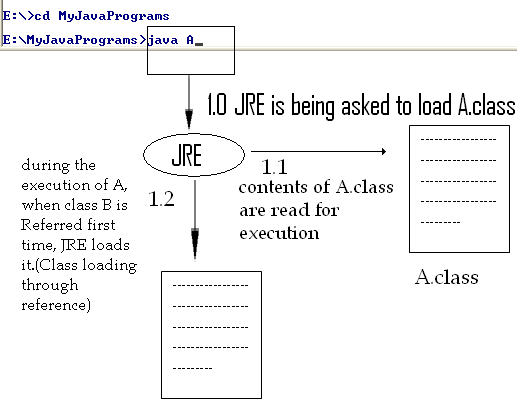
}

Static initialize is executed only once, just after a class is loaded.

*In Java, a class can be loaded in either of the following ways:-*

1. Through Explicit Introduction.
2. Through Implicit References.

In the first approach class to be loaded is explicitly introduced to the JRE.



Only a single class is loaded in an application using explicit introduction. Rest of the classes are loaded through implicit reference.

When a reference of a class is encountered first time in an already loaded class then JRE implicitly loads it before performing the operation represented by the class reference.

**class** A

{

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

{

B x = **new** B();//Reference of B is contained in A

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

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

}

}

**class** B

{

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

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

}

*At the time of class loading following sequence of steps is performed by* **JRE***:-*

1. Static Data Members (if defined) are created in the Class Area.
2. Static Initializer Block (if defined) is executed.
3. Reference of the class which resulted in class loading is resolved i.e. operation represented by the reference is performed.

**class** A

{

**static**

{

System.out.println(“A is loaded.”);

}

public A()

{

System.out.println(“A is instantiated.”);

}

}

class B

{

**static** int b;

**static**

{

b=5;

System.out.println(“B is loaded.”);

}

}

class C

{

**static**

{

System.out.println(“C is loaded.”);

}

public static void display( )

{

System.out.println(“Display( ) of C is invoked.”);

}

}

class D

{

**static**

{

System.out.println(“D is loaded.”);

}

public static void main(String args[])

{

System.out.println(“Instantiating A………”);

A x = new A();

System.out.println(“Referencing static data member b of B……”);

System.out.println(“b of B is : “ + B.b);

System.out.println(“Invoking static method of C……”);

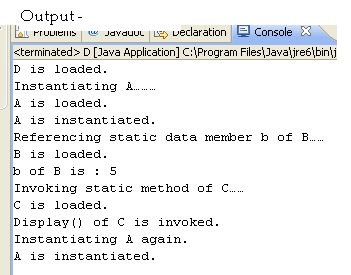
C.display();

System.out.println(“Instantiating A again.”);

A y = new A();

}

}



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

class E

{

s**tatic int a = 5;**

public static void main(String args[])

{

System.out.println(“a = “ + a);

}

}

**After compilation**

class E

{

s**tatic int a;**

static

{

a = 5;

}

----

----

}

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

class E

{

**int a = 5, b=6;**

public static void main(String args[])

{

E x = new E();

System.out.println(“a = “ + x.a);

System.out.println(“b = “ + x.b);

}

}

**After compilation**

class E

{

**int a, b;**

E( )

{

a = 5;

b = 6;

}

----

----

}

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

**class** E

{

int a=5, b=6;

public E ()

{

System.out.println(“Default.”);

}

public E(int x)

{

a=x;

System.out.println(“One parameterized.”);

}

public E(int x, int y)

{

b=y;

System.out.println(“Two parameterized.”);

}

public void display()

{

System.out.println(“a = “ + a);

System.out.println(“b = “ + b);

}

public static void main(String args[])

{

E x = new E();

E y = new E(10);

E z = new E(40, 50);

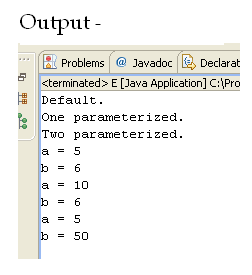
x.display();

y.display();

z.display();

}

}



**After compilation,**

**the compiler will do the following for the same program**

**class** E

{

int a, b;

public E()

{

a=5, b=6;

System.out.println(“Default.”);

}

public E(int x)

{

a=5, b=6;

a=x;

System.out.println(“One parameterized.”);

}

public E(int x, int y)

{

a=5, b=6;

b=y;

System.out.println(“Two parameterized.”);

}

public void display()

{

System.out.println(“a = “ + a);

System.out.println(“b = “ + b);

}

public static void main(String args[])

{

E x = new E();

E y = new E(10);

E z = new E(40, 50);

x.display();

y.display();

z.display();

}

}

* ***Limitations of static methods & static block:-***
  + Only static data members of a class can be referred in a static block or method.
  + A static block or static method can directly invoke only static methods.
  + ‘this’ and ‘super’ keyword cannot be used in a static method or in a static block.

*Program below executes without* **‘main’** *method –*

class Test

{

**static**

{

System.out.println(“It is executing without main…..”);

System.exit(0);

}

}

*Not only we can do this much but also we can do list of things. See the program below : -*

**class Test2**

**{**

**int a;**

**public Test2(int x)**

**{**

**a = x;**

**}**

**public void display1()**

**{**

**System.out.println("a = " + a);**

**}**

**static**

**{**

**System.out.println("Test is loaded.");**

**Test2 t = new Test2(5);**

**t.display1();**

**P x = new P();**

**System.exit(0);**

**}**

**}**

**class P**

**{**

**public P()**

**{**

**System.out.println("P is instantiated.");**

**}**

**static**

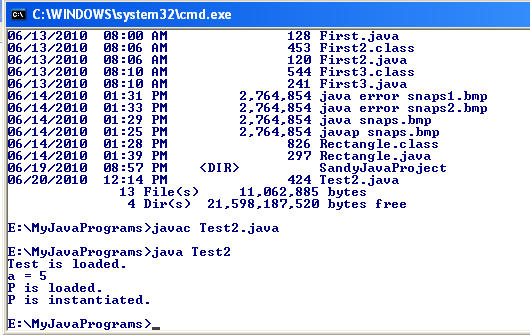
**{**

**System.out.println("P is loaded.");**

**}**

**}**

**OUTPUT -**



Now, as we saw that we can do almost everything without having **main( )**, then **why we need main( )** **in our program?**

*In the sessions ahead, we will get the answer.*

Passing arguments to methods-

In Java, primitive type arguments are passed to methods by value i.e. their copy is created in the invoked method.

Objects are passed by references i.e. in case of objects, copy of their reference variables is created in the invoked method.

Now the program below is written to swap the values.

*Observe it carefully to understand how this issue is being resolved.*

**public** **class** Swapper

{

**public** **static** **void** swap(**int** x, **int** y)

{

**int** z;

z = x;

x = y;

y = z;

}

}

**public** **class** SwapperTest

{

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

{

**int** a =5, b = 6;

System.*out*.println("a = " + a);

System.*out*.println("b = " + b);

Swapper.*swap*(a, b);

System.*out*.println("After swap");

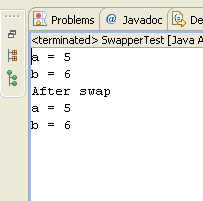
System.*out*.println("a = " + a);

System.*out*.println("b = " + b);

}

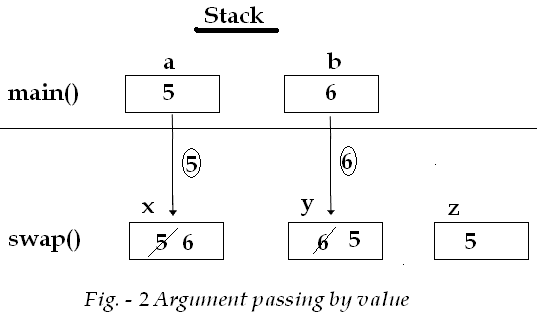
}

**The output is -**



Our purpose is not solved. Let’s see the reason for this.

See the fig2. below



Now, let’s find out solution to this problem.

**public** **class** MyNumber

{

**int** value;

**public** MyNumber(**int** x)

{

value = x;

}

}

**public** **class** Swapper

{

**public** **static** **void** swap(MyNumber x, MyNumber y)

{

**int** z;

z = x.value;

x.value = y.value;

y.value = z;

}

}

**public** **class** SwapperTest

{

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

{

MyNumber a = **new** MyNumber(5);

MyNumber b = **new** MyNumber(6);

System.*out*.println("a = " + a.value);

System.*out*.println("b = " + b.value);

Swapper.*swap*(a, b);

System.*out*.println("After swap");

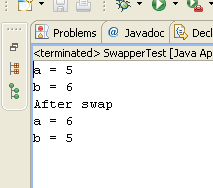
System.*out*.println("a = " + a.value);

System.*out*.println("b = " + b.value);

}

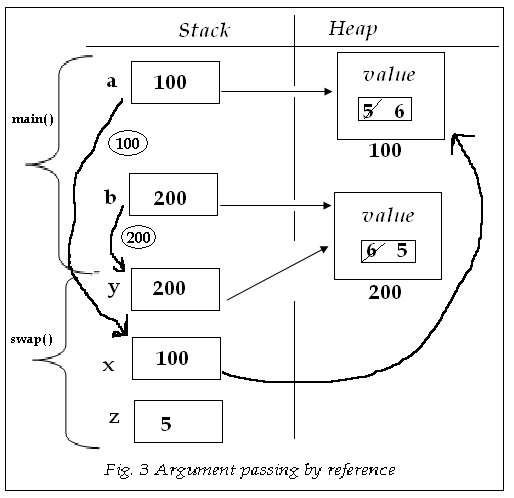
}

Output –



Now the problem is fixed as you can see the value is swapped. We handled it by passing value by reference.

Let see the explanation diagrammatically. (Fig. - 3)



**Question**. Define a class named Rational that contains **2 data members** to store the value of numerator and denominator of a rational number, **default** & **two parameterized constructors**, a **display ( ) method** that displays the value of a rational object in form and **add ( ) methods** which are referenced by the following class.

**Answer** –

**package** StaticConceptualPrograms;

**public** **class** Rational

{

**int** numerator, denominator;

**public** Rational() { }

**public** Rational(**int** x, **int** y)

{

numerator = x;

denominator = y;

}

**void** display()

{

System.*out*.println(numerator+"/"+denominator);

}

Rational add(Rational y)

{

numerator = y.denominator\***this**.numerator + **this**.denominator \* y.numerator;

denominator = y.denominator\***this**.denominator;

**return** **this**;

}

Rational add(Rational x, Rational y)

{

numerator = y.denominator\*x.numerator + x.denominator \* y.numerator;

denominator = y.denominator\*x.denominator;

**return** **this**;

}

/\*

static Rational add(Rational x, Rational y)

{

}

\*/

}

**package** StaticConceptualPrograms;

**public** **class** RationalTest

{

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

{

Rational a = **new** Rational(2, 3);

Rational b = **new** Rational(4, 5);

System.*out*.print("Rational a is : ");

a.display();

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

System.*out*.print("Rational b is : ");

b.display();

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

Rational c = a.add(b);

System.*out*.print("Sum of a & b : ");

c.display();

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

Rational r = **new** Rational(6, 5);

Rational s = **new** Rational(4, 3);

Rational t = **new** Rational();

t.add(r, s);

System.*out*.print("Rational r is : ");

r.display();

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

System.*out*.print("Rational s is : ");

s.display();

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

System.*out*.print("Rational r & s is : ");

t.display();

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

Rational z;

System.*out*.print("Sum of a, b, r & s is : ");

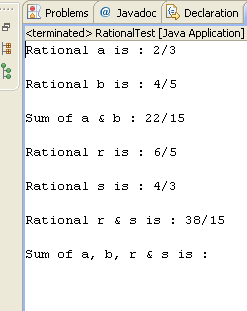
//z = Rational.add(c, t);

//z.display();

}

}

**Output -**



**public** **class** Rational

{

**int** p, q;

**public** Rational() {}

**public** Rational(**int** x, **int** y)

{

p = x;

q = y;

}

**public** **void** display()

{

System.*out*.println(p+"/"+q);

}

Rational add(Rational r)

{

Rational s = **new** Rational();

s.p = p \* r.q + q \* r.p;

s.q = q \* r.q;

**return** s;

}

**public** **void** add(Rational a, Rational b)

{

p = a.p \* b.q + a.q \* b.q;

q = a.q \* b.q;

}

**public** **static** Rational add(Rational a, Rational b)

{

Rational c = **new** Rational();

c.p = a.p \* b.q + a.q \* b.q;

c.q = a.q \* b.q;

**return** c;

}

}

**public** **class** RationalTest

{

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

{

Rational a = **new** Rational(2, 3);

Rational b = **new** Rational(4, 5);

System.*out*.print("Rational a is : ");

a.display();

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

System.*out*.print("Rational b is : ");

b.display();

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

Rational c = a.add(b);

System.*out*.print("Sum of a & b : ");

c.display();

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

Rational r = **new** Rational(6, 5);

Rational s = **new** Rational(4, 3);

Rational t = **new** Rational();

t.add(r, s);

System.*out*.print("Rational r is : ");

r.display();

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

System.*out*.print("Rational s is : ");

s.display();

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

System.*out*.print("Rational r & s is : ");

t.display();

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

Rational z;

System.*out*.print("Sum of a, b, r & s is : ");

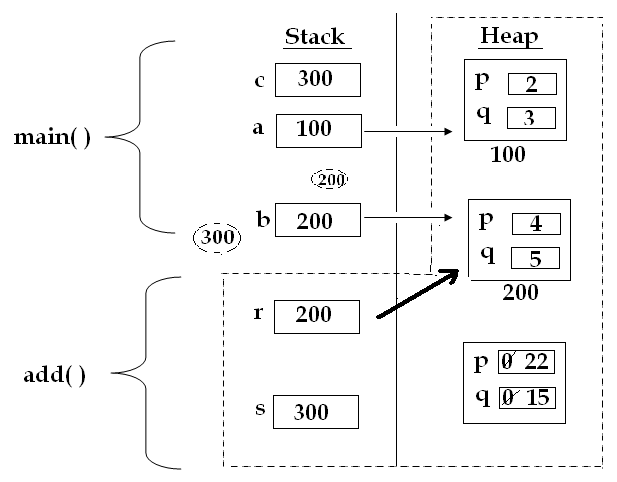
z = Rational.add(c, t);

z.display();

}

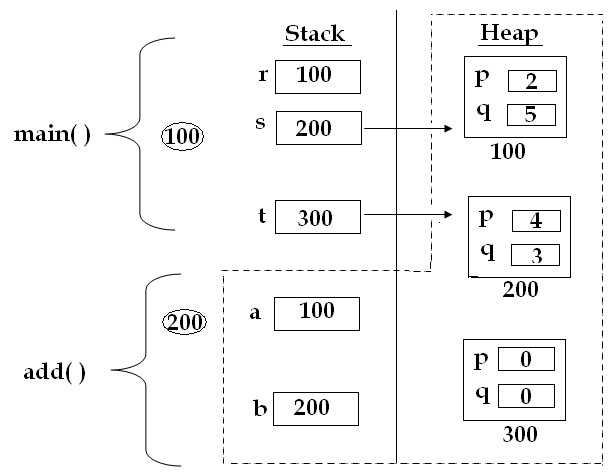
}

To understand the logic used inside the different methods of **Rational** class, please see the diagrams below.

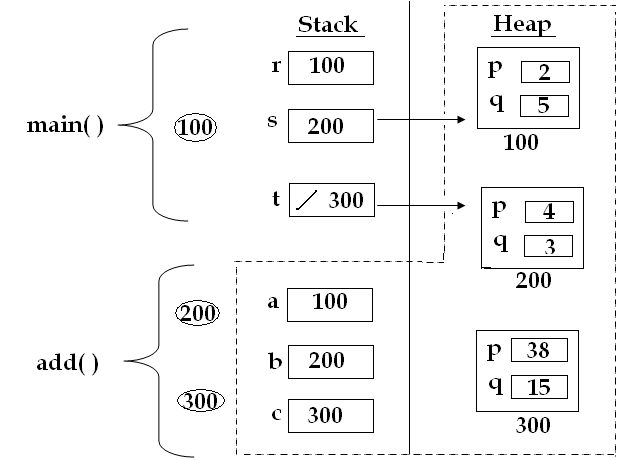


The dotted line here indicates the add( ) method’s scope.

Figure (20.a) to understand **public Rational add(Rational r) method**



The dotted line here indicates the add( ) method’s scope.



The dotted line here indicates the add( ) method’s scope.

**Program -**

**class** Test1

{

**int** a, b;

**public** Test1(**int** a, **int** b)

{

a=a;

b=b;

}

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

{

Test1 t = **new** Test1(5, 6);

t.display();

}

**public** **void** display()

{

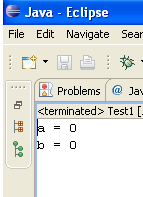
System.*out*.println("a = " + a);

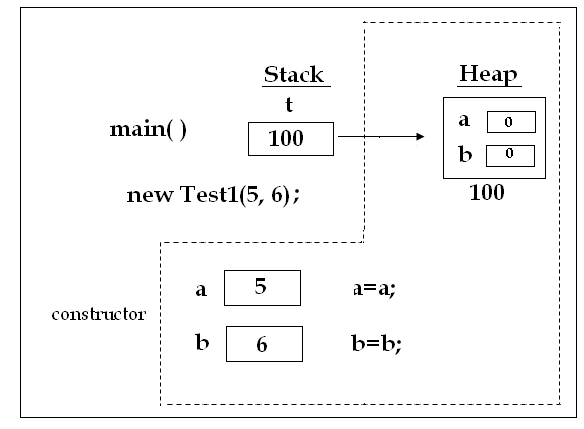
System.*out*.println("b = " + b);

}

}

**Output -**





**this** keyword-

In Java, each non-static method & constructor has an implicit parameter named **this** which holds the reference of the invoking object.

Each non static method and constructor has **this** as its first parameter.

*Program* **Test1** *as understood by* **JRE** *is rewritten below:-*

**class** Test1

{

**int** a, b;

Each non static method and constructor has **this** as its first parameter.

**public** Test1(**Test1 this, int** a, **int** b)

{

a=a;

b=b;

}

**public** **void** display(**Test1 this**)

{

System.*out*.println("a = " + this.a);

System.*out*.println("b = " + this.b);

}

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

{

Test1 t = **new** Test1(5, 6);

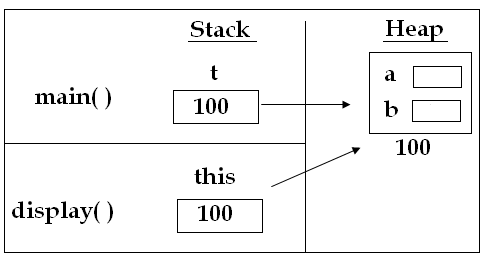
t.display();

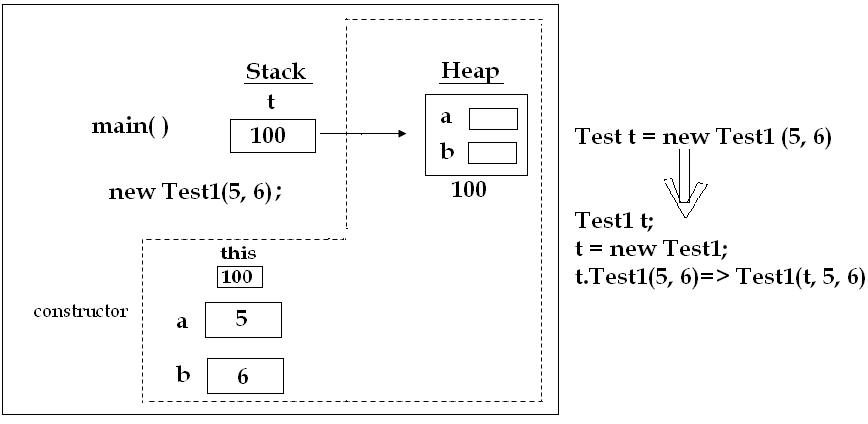
}

}

As understood by JRE

t.display( ); display(t);





The dotted line here indicates the constructors’ scope.

1. **First usage of ‘this’:-**

* ‘**this’** keyword is used to identify data members of invoking objects in a method or constructor. In case, there is a conflict between object data member and local variables we use –

Syntax:-

this.<memberName>

**Example Program -**

**class** Test1

{

**int** a, b;

**public** Test1(**int** a, **int** b)

{

**this**.a=a;

**this**.b=b;

}

**public** **void** display()

{

System.*out*.println("a = " + **this**.a);

System.*out*.println("b = " + **this**.b);

}

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

{

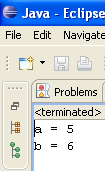
Test1 t = **new** Test1(5, 6);

t.display();

}

}

**Output -**



1. **Second usage of ‘this’:-**

* ‘this’ keyword facilitate chaining of constructors of a class.
  + **Constructor Chaining** is the facility in which one constructor of a class invokes another constructor of the same class.

**Syntax:-**

**this(arguments if any)**

NOTE: When ‘this’ keyword is used to invokes a constructor from another constructor, it must be the first statement in the invoking constructor.

**Example Program -**

**public** **class** ThisTestInMethod

{

**int** p, q;

**public** ThisTestInMethod()

{

**this**(2, 3);

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

}

**public** ThisTestInMethod(**int** x)

{

**this**(x, 3);

System.*out*.println("One Parameterized Constructor...");

}

**public** ThisTestInMethod(**int** x, **int** y)

{

p = x;

q = y;

System.*out*.println("Two Parameterized Constructor...");

}

**public** **void** display()

{

System.*out*.println("p = " + p);

System.*out*.println("q = " + q);

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

}

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

{

ThisTestInMethod a = **new** ThisTestInMethod();

a.display();

ThisTestInMethod b = **new** ThisTestInMethod(10);

b.display();

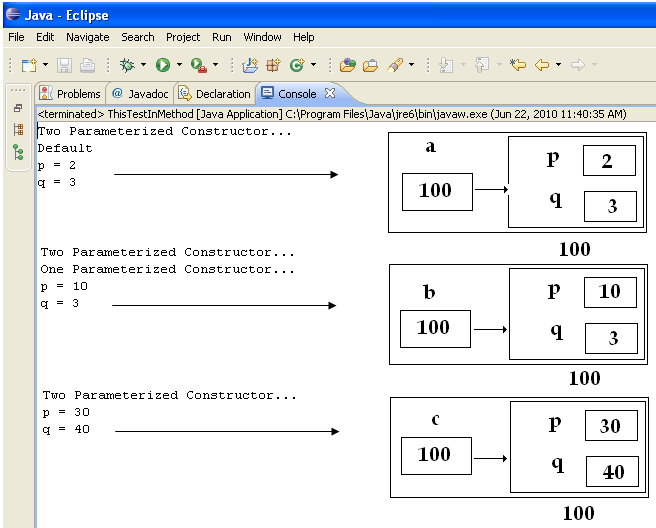
ThisTestInMethod c = **new** ThisTestInMethod(30, 40);

c.display();

}

}

**Output –**



1. **Third usage of ‘this’:-**

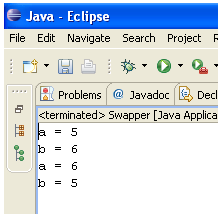
* ‘this’ keyword facilitate method chaining.
  + **Method Chaining** is the facility of invoking multiple methods on an object in a single statement.

*Program* **(Swapper)** *to understand the 3rd usage of* ‘this’*. This is a Swapper program which has been written without using* ‘this’ *keyword.*

*Then we will see how efficiently we can write the same program using* ‘this’ *keyword.*

1. **public** **class** Swapper
2. {
3. **int** a, b;
5. **public** Swapper(**int** x, **int** y)
6. {
7. a = x;
8. b = y;
9. }
11. **public** **void** swap()
12. {
13. **int** c = a;
14. a = b;
15. b = c;
16. }
18. **public** **void** display()
19. {
20. System.*out*.println("a = " + a);
21. System.*out*.println("b = " + b);
22. }
24. **public** **static** **void** main(String[] args)
25. {
26. Swapper x = **new** Swapper(5, 6);
27. x.display();
28. x.swap();
29. x.display();
30. }
31. }

**Output –**

**

*Now, in the same program* **(Swapper)***, we will do some modifications to understand the 3rd usage of* ‘this’*keyword.*

*The same above* **Swapper** *program which has been written using* ‘this’ *keyword.*

**public** **class** Swapper

{

**int** a, b;

**public** Swapper(**int** x, **int** y) //Constructor implicitly return ‘**this’**

{

a = x;

b = y;

}

**public** Swapper swap()

The changes being made in these places indicated here.

{

**int** c = a;

a = b;

b = c;

**return** **this**;

}

**public** Swapper display()

{

System.*out*.println("a = " + a);

System.*out*.println("b = " + b);

**return** **this**;

}

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

{

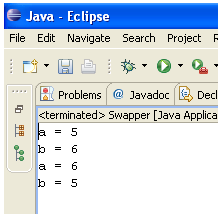
Swapper x = **new** Swapper(5, 6);

x.display().swap().display(); // Method calls are chained.

}

}

**Output -**

**

*Now, in the same program we will again do some more modifications in the* ***main*** *method. Observe it carefully.*

**public** **class** Swapper

{

**int** a, b;

**public** Swapper(**int** x, **int** y) //Constructor implicitly return ‘**this’**

{

a = x;

b = y;

}

**public** Swapper swap()

The changes being made again only in **main** method.

{

**int** c = a;

a = b;

b = c;

**return** **this**;

}

**public** Swapper display()

{

System.*out*.println("a = " + a);

System.*out*.println("b = " + b);

**return** **this**;

}

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

{

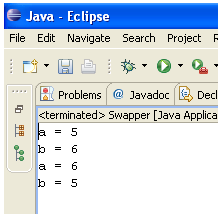
new Swapper(5, 6).display().swap().display(); // Method calls

are chained.

}

}

**Output -**

**

NOTE : All constructors return ‘**this’**.

**Association** : - When two classes are related in some way they are said to be associated. Relation between classes of two types:-

1. Is – A Relation
2. Has – A Relation

**Is – A Relation** between classes is implemented with the help of inheritance.

**Question: What is Inheritance?**

Inheritance is the process of extending the functionality of a class by defining a new class which inherits all the features of the existing class and adds some features of its own. Inheritance is a means of implementing generalization.

**Question: Why Inheritance is used?**

Inheritance is used for code Reusability and runtime polymorphism.

**extends -** In java ‘extends’ keyword is used to inherit the features of one class into another.

Syntax:-

this(arguments if any)

**Syntax :-**

**class Identifier extends BaseClassName**

**{**

**Additional methods**

**Or**

**Additional dataMembers & methods**

**}**

Program to understand the working of Inheritance: -

**public** **class** Common

{

**int** l, b;

**public** Common(**int** x, **int** y)

{

l = x;

b = y;

}

**public** **void** display()

{

System.*out*.println("Length = " + l);

System.*out*.println("Breadth= " + b);

}

}

**public** **class** Rect **extends** Common

{

**public** Rect(**int** x, **int** y)

{

**super**(x, y);

}

**public** **int** area()

{

**return** l \* b;

}

}

**public** **class** Cuboid **extends** Common

{

**int** h;

**public** Cuboid(**int** x, **int** y, **int** z)

{

**super**(x, y);

h = z;

}

**public** **void** display()

{

**super**.display();

System.*out*.println("Height = " + h);

}

**public** **int** volume()

{

**return** l \* b \* h;

}

}

**public** **class** InheritanceTest

{

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

{

Rect r = **new** Rect(5, 4);

Cuboid c = **new** Cuboid(6, 5, 8);

System.*out*.println("Dimension of rectangle : ");

r.display();

System.*out*.println("Dimension of cuboid : ");

c.display();

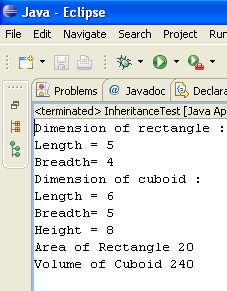
System.*out*.println("Area of Rectangle " + r.area());

System.*out*.println("Volume of Cuboid " + c.volume());

}

}

**Output –**

****

NOTE :

1. Constructor of a class is never inherited in another class.
2. Inheritance is always unidirectional i.e. child class knows everything of parent class but the reverse is not true.

**Usage of ‘super’ keyword**

1. ‘super’ keyword is used to chain superclass and subclass constructors, i.e. to invoke a superclass constructor form a subclass constructor.

Syntax –

super(Argument if any);

NOTE: ‘super’ when used in subclass constructor to invoke superclass constructor than it must be the first statement.

1. ‘super’ keyword is used to invoke a superclass method from a subclass method.

Syntax –

super.MethodName( );

1. ‘super’ keyword is used to refer superclass data members in a subclass, in case there is a name conflict between superclass and subclass data members.

Syntax –

super.MemberName;

**Method Overriding: -**If a class defines a method of same signature as a method of its superclass then the class is said to be overriding method of the superclass.

Method overriding is one of the means of implementing polymorphism.

**HOMEWORK** –

**Assignment 1. - What is runtime polymorphism in Java?**

**Polymorphism** is the capability of an action or method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementation. This is one of the basic principles of object oriented programming.

The *method overriding* is an example of **runtime polymorphism**. You can have a method in subclass overrides the method in its super classes with the same name and signature. Java virtual machine determines the proper method to call at the runtime, not at the compile time.

Let's take a look at the following example:

**PROGRAM** RuntimePolymorphismDemo.java

class Animal {

void whoAmI() {

System.out.println("I am a generic Animal.");

}  
}  
class Dog extends Animal {

void whoAmI() {

System.out.println("I am a Dog.");

}  
}  
class Cow extends Animal {

void whoAmI() {

System.out.println("I am a Cow.");

}  
}  
class Snake extends Animal {

void whoAmI() {

System.out.println("I am a Snake.");

}  
}  
  
class RuntimePolymorphismDemo {

public static void main(String[] args) {

Animal ref1 = new Animal();

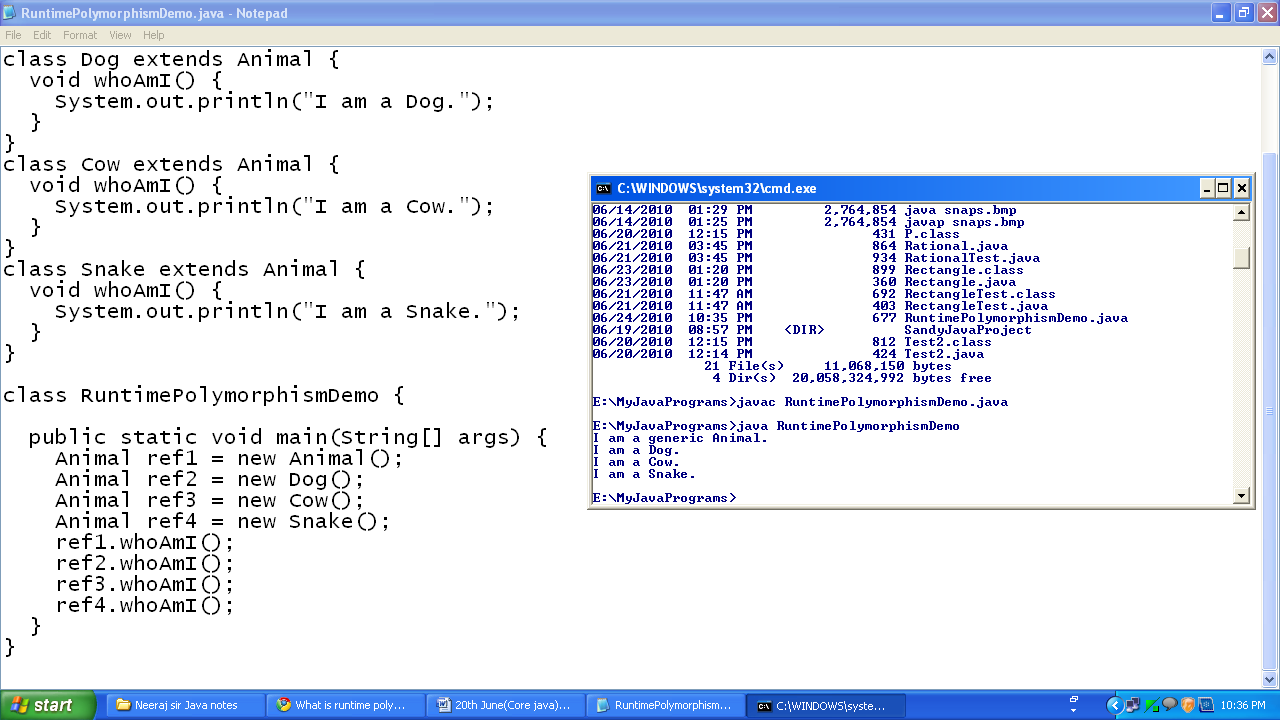
Animal ref2 = new Dog();

Animal ref3 = new Cow();

Animal ref4 = new Snake();

ref1.whoAmI();  
 ref2.whoAmI();  
 ref3.whoAmI();  
 ref4.whoAmI();  
 }  
}

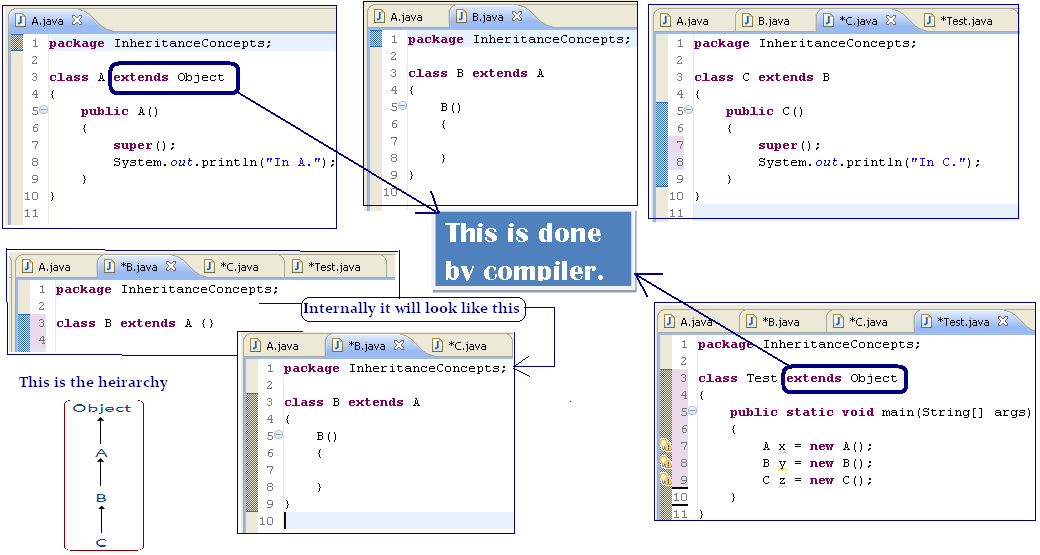
**THE OUTPUT IS-**



In the example, there are four variables of type Animal (e.g., *ref1*, *ref2*, *ref3*, and *ref4*). Only *ref1* refers to an instance of *Animal* class, all others refer to an instance of the subclasses of *Animal*. From the output results, we can confirm that version of a method is invoked based on the actually object's type.

In Java, a variable declared type of class *A* can hold a reference to an object of class *A* or an object belonging to any subclasses of class *A*. The program is able to resolve the correct method related to the subclass object at runtime. This is called the runtime polymorphism in Java. This provides the ability to override functionality already available in the class hierarchy tree. At runtime, which version of the method will be invoked is based on the type of actual object stored in that reference variable and not on the type of the reference variable.

**Date : 26.06.10**

****

In Java, each class is direct or indirect subclass of Object. Relation between Object class and other classes is implicitly created by the compiler at the time of compilation.

Whenever object of a class is created, Object class must be given a chance to initialize its part of object state.

*Invocation of object constructor is determined by the compiler by performing following steps at the time of compilation-*

1. If a class doesn’t define any constructor then compiler defines a constructor & writes super to it.
2. If a class contains constructor compiler simply adds super keyword to them.

**See the example below -**

**class** A

{

**public** A()

{

System.*out*.println("In A.");

}

}

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

{

**public** B()

{

System.*out*.println("In B default.");

}

**public** B(**int** x)

{

System.*out*.println("In B.");

}

}

**class** C **extends** B

{

**public** C()

{

System.*out*.println("In C.");

}

}

**class** Test **extends** Object

{

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

{

A x = **new** A();

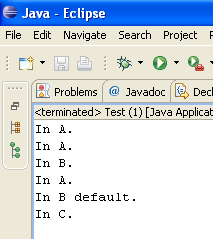
B y = **new** B(5);

C z = **new** C();

}

}

Output -



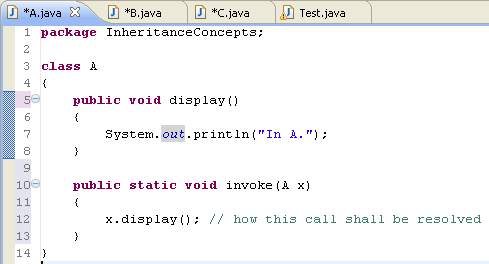
**NOTE** : If a class contains only parenthesized constructors then its subclass must explicitly invoke then using ‘**super’**.

**DYNAMIC BINDING AND RUNTIME POLYMORPHISM**

Resolving a method call i.e. finding out a method definition to be executed for a method call is called binding.

If a method call is resolved at compilation time it is called Static Binding or Early Binding.

If a method call is resolved at the time of execution it is called Late Binding or Dynamic Binding.



In Java, method calls are resolved according to the following rules:-

1. Static methods are statically binded.
2. Non-static methods are dynamically binded except following 3 cases:-
   * 1. Private Non-static methods are statically binded.
     2. Constructors are statically binded.
     3. Methods calls made using *super* keywords are statically binded.

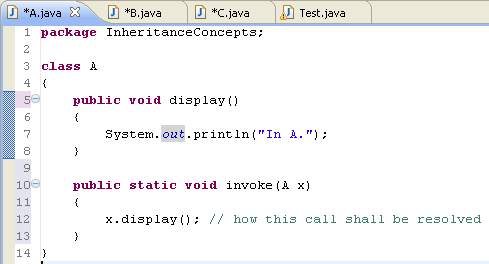
In JVM assembly there are 4 instructors which are used by JRE invoking methods.

|  |  |  |
| --- | --- | --- |
| **JVM Assembly** | **Purpose** | **Type of Binding** |
| 1. **invokestatic** | Used to invoke static methods | static binding |
| 1. **invokespecial** | Used to invoke Constructors, private non-static methods & non-private, non-static methods using super keyword. | static binding |
| 1. **invokevirtual** | Used to invoke non-private, non-static methods without super. | dynamic binding |
| 1. **invokeinterface** | Used to invoke interface methods. | dynamic binding |

**CONCLUSION**

* static method static binding
* Constructor / private instance method / non-private method using super static binding
* Non-private instance method dynamic binding
* To invoke interface method dynamic binding

Now the solution for the problem



**class** A

{

**public** **void** display()

{

System.*out*.println("In A.");

}

**public** **static** **void** invoke(A x)

{

x.display(); // how this call shall be resolved

}

}

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

{

**public** **void** display()

{

**super**.display();

System.*out*.println("In B.");

}

}

**class** C **extends** B

{

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

{

A x = **new** A();

B y = **new** B();

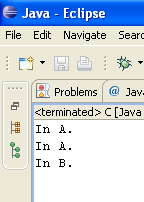
A.*invoke*(x);

A.*invoke*(y);

}

}

OUTPUT –



**“ BEHAVIOR ”**

**IS ASSOCIATED WITH**

**Class**

**Object**

**Family of classes having common trend**

**IS ASSOCIATED WITH**

When a class extends another class then apart from the features of the super class subclass also inherits the name of its superclass i.e. reference variable of a superclass can be used to refer subclass objects.

Let there be following class hierarchy –

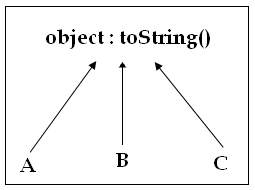
|  |  |
| --- | --- |
| **REFERENCE TYPE** | **OBJECT TYPE** |
| Object, A | A |
| Object, A, B | B |
| Object, A, B, C | C |

**Date : 27.06.10**

**Common Behaviour** – In a family of classes can be supported in the following two ways-

1. **By Gift** – Parent Class defines methods that are to be provided to all subclasses.

e.g. :-

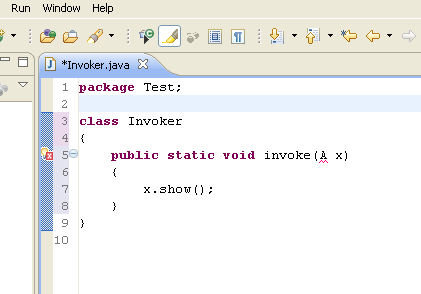


1. **By force** – In this approach all subclasses of a family are forced which provide implementation of the common behavior of the family.
   1. **abstract** keyword is used in the implementation of this strategy. **abstract** keyword is used to define abstract methods & classes.
   2. An **abstract** method is a method without implementation which represents what is to be done without specifying how it could be done.
   3. If a class contains any abstract method then class is declared as “abstract”. An abstract class has following characteristics:-
      1. It cannot be instantiated.
      2. It imposes the responsibility of providing implementation of all its abstract methods on its subclasses.
      3. If any of its subclass fails to define even a single abstract method of the superclass then subclass is also declared as “abstract”.

**Q1. Problem Statement:-**

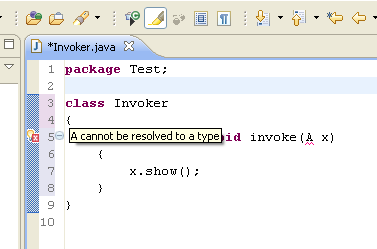
Define a class name Invoker that contains a public static void method name invoke(). In this method an object of any subclass of class A is passed as argument. You are required to invoke a method name show() on this parameter object from the invoke().

**Solution** –



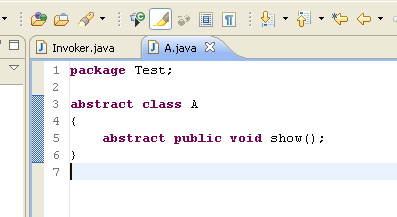
The problem occurs here is follows : -

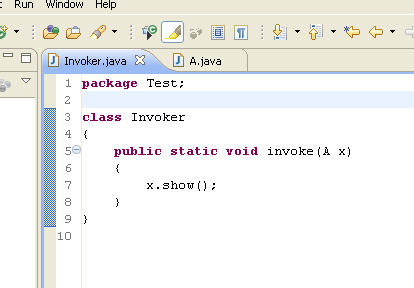
1. class A is not defined.



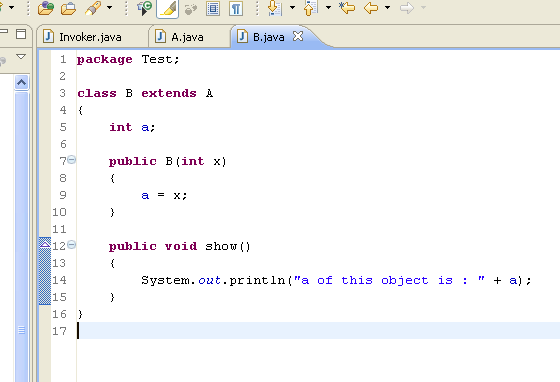
Now as soon as we created a abstract class A, the error message went away.

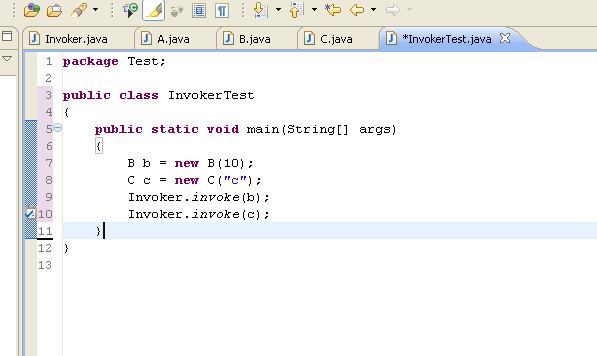
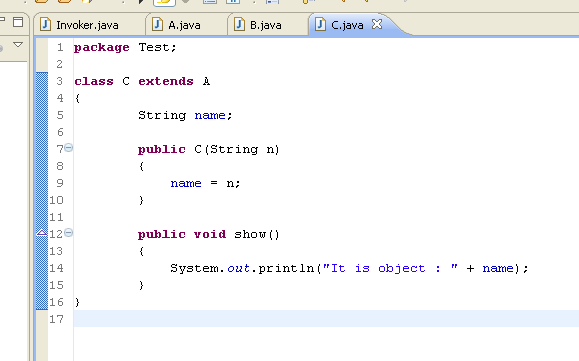
See the figures below –



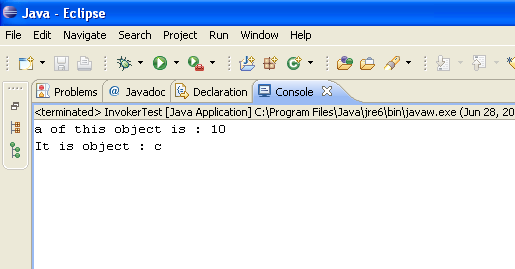


Again we need to complete our Problem Statement.





**THE OUTPUT IS -**



**class** Invoker

{

**public** **static** **void** invoke(A x)

{

x.show();

}

}

**abstract** **class** A

{

**abstract** **public** **void** show();

}

**abstract** **class** A

{

**abstract** **public** **void** show();

}

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

{

**int** a;

**public** B(**int** x)

{

a = x;

}

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

{

System.*out*.println("a of this object is : " + a);

}

}

**class** C **extends** A

{

String name;

**public** C(String n)

{

name = n;

}

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

{

System.*out*.println("It is object : " + name);

}

}

**public** **class** InvokerTest

{

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

{

B b = **new** B(10);

C c = **new** C("c");

Invoker.*invoke*(b);

Invoker.*invoke*(c);

}

}

**Interface** : - An interface is a collection of implicit abstract methods and static final data members. Interfaces are used to abstract the interface of the classes from their implementation.

*Syntax of defining an interface:-*

interface Identifier

{

static final dataMembers

implicit abstract methods

}

Or

interface Identifier

{

static final dataMembers

}

Or

interface Identifier

{

implicit abstract methods

}

**Example –**

interface Printable

{

void print();

}

Interfaces are implemented by classes.

*Syntax of implementing an interface:-*

class className implements InterfaceName

{

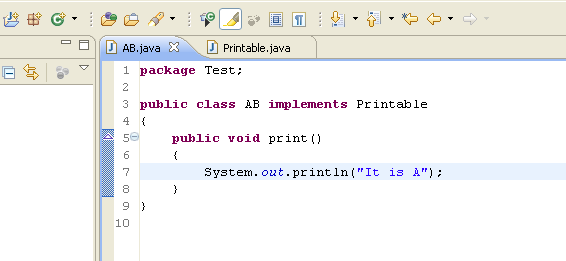
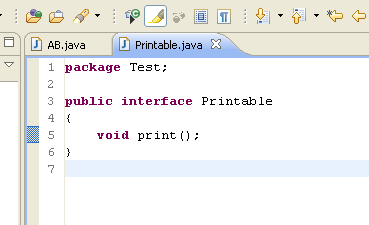
public definition of interface methods

addition members (if any) of the class

}

**NOTE**: If a class implements an interface then it has to provide public definition of all interface methods otherwise the class is declared as abstract.

Example :

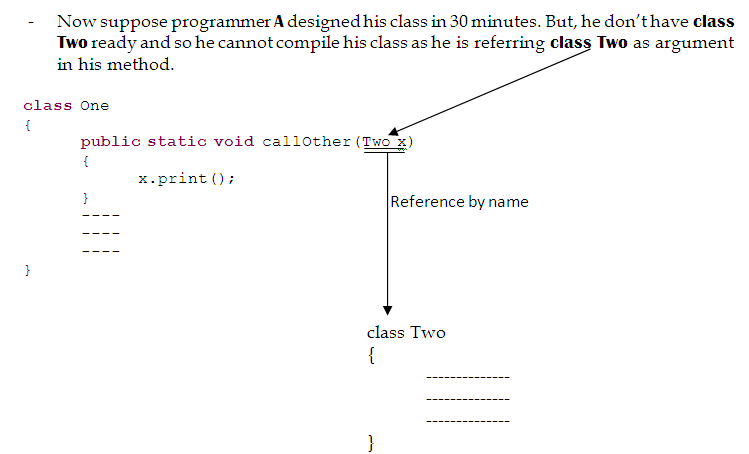


|  |  |  |
| --- | --- | --- |
|  | **Class** | **Interface** |
| 1. **Degree of abstraction** | Classes support 0 – 100% abstraction. | Interfaces support only 100% abstraction. |
| 1. **Type of Inheritance** | Classes facilitate implementation inheritance. In Java, multiple implementation inheritance is not allowed. | Interfaces facilitate interface inheritance. In Java, multiple interface inheritance is allowed. |
| 1. **Type of classing environment** | Classes facilitate static classing environment & dynamic classing environment only within a family. | Interfaces facilitate dynamic classing environment across multiple families. |

|  |  |  |
| --- | --- | --- |
| **Type of Inheritance** | **Purpose** | **Anlogy(Similarity)** |
| 1. Implementation Inheritance | Code reusability, Runtime polymorphism. | Represents Blood relation of real life.  E.g. – A is son of B. |
| 1. **Type of Inheritance** | * Runtime polymorphism. * ‘implements’ keyword is used. | Represents Non-Blood relation of real life.  E.g. –  B is friend of C.  C is a teacher. |

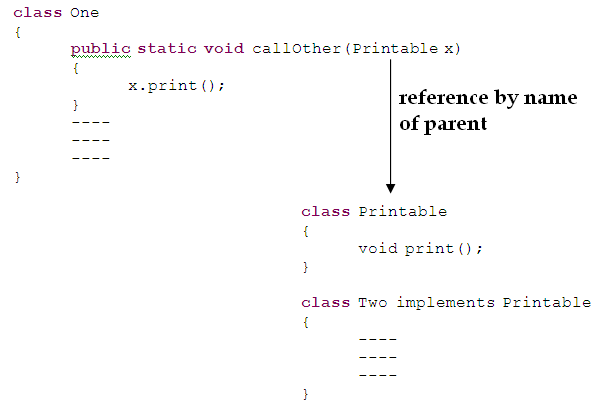
*Literal meaning of Interface is* ‘**medium between two things**’.

* Let there be two programmer named A and B who are defining classes One and Two respectively. A need to refer class of B in his class. class One is simple, A would take at most 30 minutes to complete it. class Two is complex and B would require at least 10 days to complete it.



*A class can be referred in another class in the following three ways:-*

1. By Name – In this approach, class is referred in another class directly by its name. This approach has following problems:-
   1. class must be known and available at the time of compiling the referencing class.
   2. Reference by name created a type coupling between referencing & referenced class which results in maintenance problem.
   3. The limitation of referencing only known and available classes is called **static classing environment**.
2. By the name of parent:-



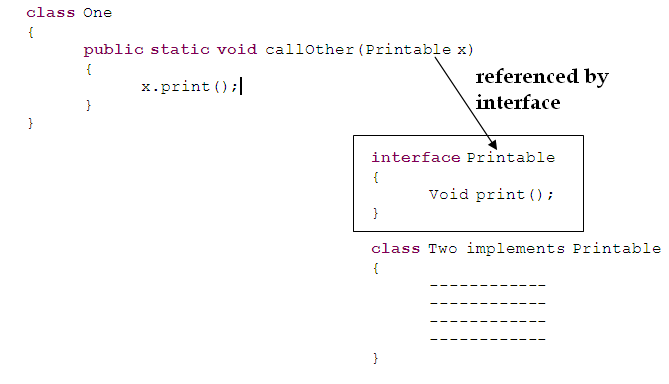
**Advantage of this approach:**

* 1. In this approach a class is referred in another class by the name of its parent or family. This approach has following advantages :
     1. Only parent class need to be known or available at the time of creating the reference.
     2. Referencing & referenced class are being coupled.
     3. Facility of referencing unknown & unavailable classes is **called dynamic classing environment**.

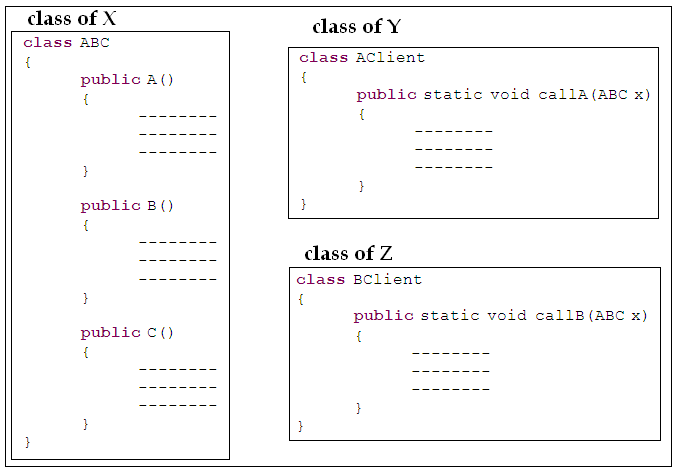
**Disadvantage of this approach:**

1. **Disadvantage of this approach** is that dynamic classing environment is supported within a family.
2. By the name of its interface:-

In this approach class is referred in another class through the interface implemented by the class.

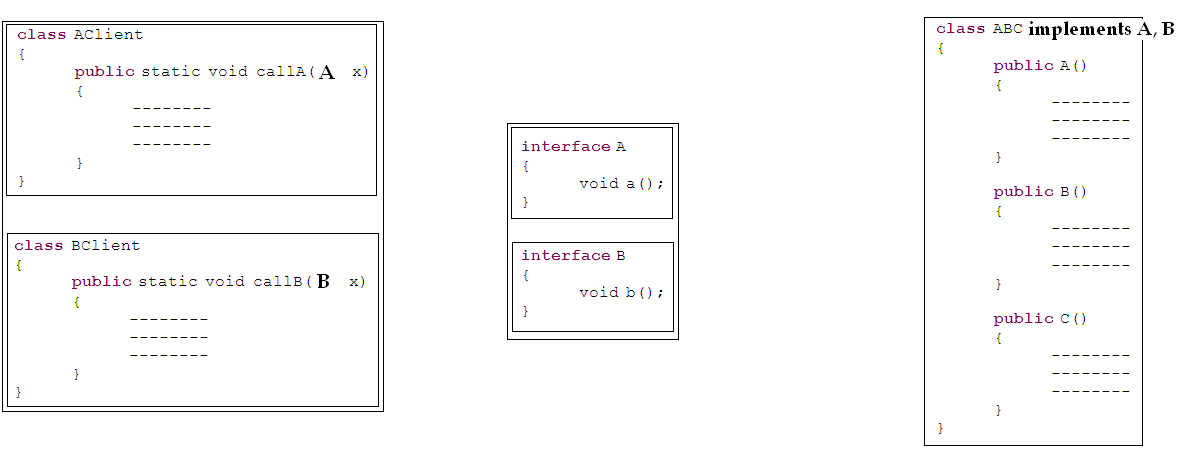


* Let there be three programmers named X, Y and Z. X is defining a class named ABC that contains a(), b() and c() methods. X wants to expose only a() method of his class to the class of Y and only b() method to the class of Z.



Now if we make some changes here, everything will go right. See the changes we will do in the figure below:-

Figure ABC



**Q2. Problem Statement:-**

Define a class name Invoker that contains a public static void method name invoke(). In this method, an object is received as argument. You are required to invoke a method named show() on the argumented object from the invoke().

Solution -

class Invoker

{

public static void invoke(Showable o)

{

o.show();

}

}

interface Showable

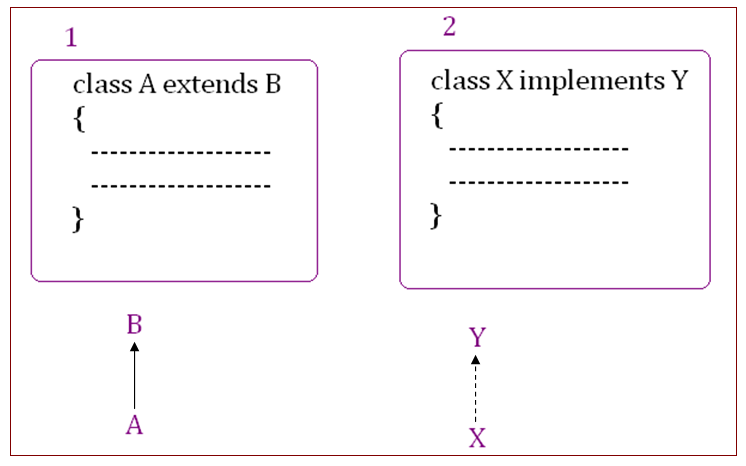
{

void show();

}

**Dated : 03.07.10**

**Is-a** relation is of 2 types –



**Has-a** Relation

* **Aggregation** represents part and whole relation between objects. In case of aggregation part and whole may have their independent existence.

*For example Room has chairs.*

* **Composition** represents a stronger **has-a** relation between objects in which existence of one object depends on another.

*For example Room has walls.*

**package** HasAIsARelation;

**class** A

{

**int** a;

**public** A(**int** x)

{

a=x;

}

**public** **void** display()

{

System.*out*.println("a = " + a);

}

}

**package** HasAIsARelation;

**class** AB

{

A a;

**int** b;

**public** AB(**int** x, **int** y)

{

a=**new** A(x);

b=y;

}

**public** **void** display()

{

a.display();

System.*out*.println("b = " + b);

}

}

**package** HasAIsARelation;

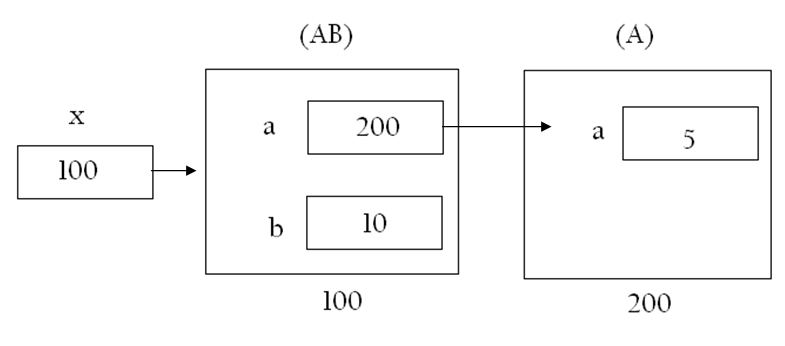
**class** HasATest

{

AB x = **new** AB(5, 10);

x.display();

}



------------------------------------------------X-----------------------------------------------

**NESTED/INNER CLASS**

A class that is defined within the scope of another class is called nested or inner class.

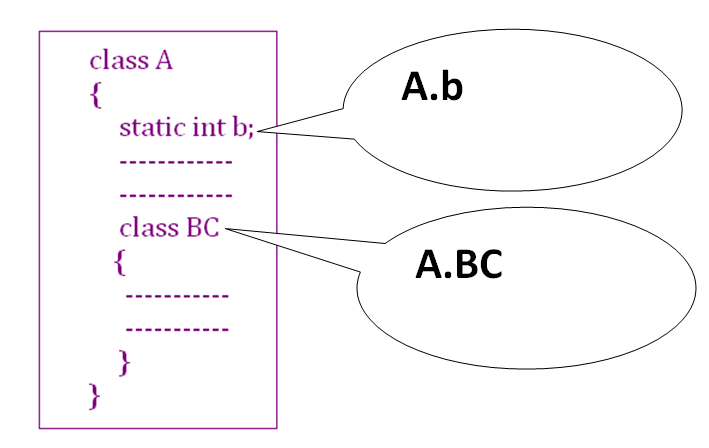
Nested classes can be of 2 types:-

1. **STATIC INNER class -** A class that is defined within the scope of another class and is qualified by static keyword is called static inner class. It has following characteristics:-
   * 1. It can be instantiated independently, i.e. an object of outer class is not required for the instantiation of this class.
     2. It can refer all static data members of its outer class irrespective of their scope.

**NOTE**: If a class is defined within the scope of another class it is referred according to the following syntax.

*outerClassName.InnerClassName*

*See the figure below -*



*Example –*

Program **InnerTest.java**

**package** InnerClassOuterClass;

**class** A

{

**private** **static** **int** *a*;

**static**

{

*a*=5;

System.*out*.println("A is loaded...");

}

// static inner class starts

**public** **static** **class** B

{

**int** b;

**public** B(**int** x)

{

b=x;

}

**public** **void** display()

{

System.*out*.println("a of outer class is : " + *a*);

System.*out*.println("b of this object is : " + b);

}

}

// static inner class ends.

}

**class** InnerTest

{

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

{

System.*out*.println("Initiating static inner class..");

A.B x = **new** A.B(10);

System.*out*.println("invoking display() method on inner

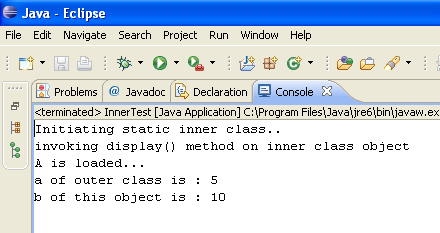
class object");

x.display();

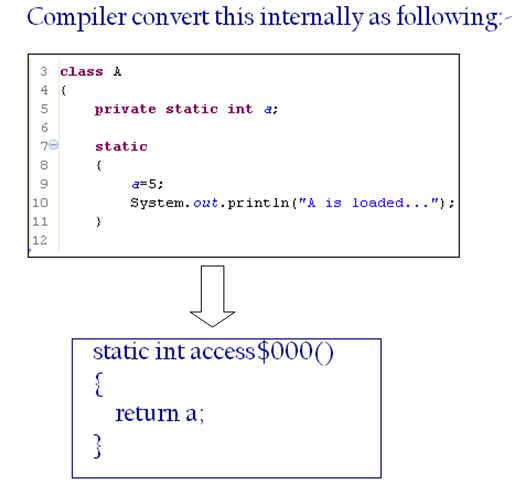
}

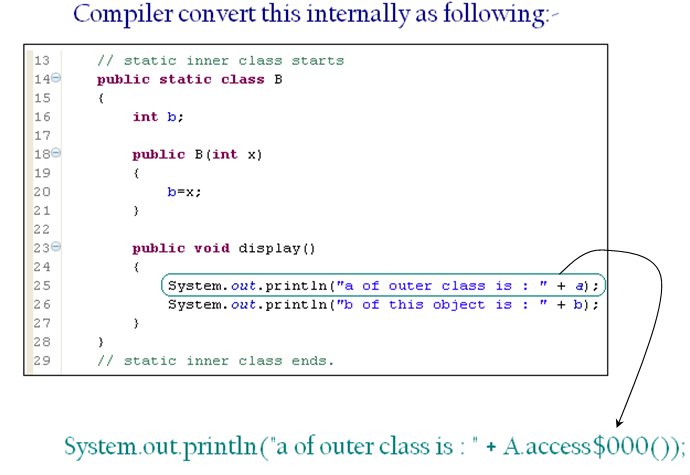
}

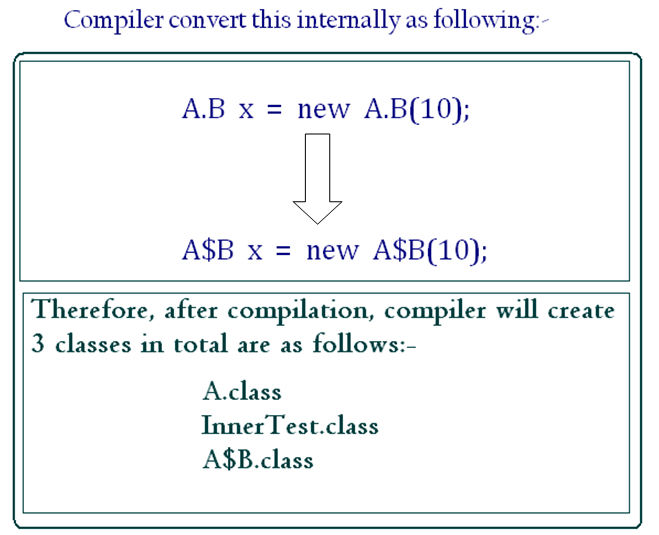
Output –



* Explanation of above program (**InnerTest.java**)







After compilation contents of inner class are contained in a separate class file which is named according to the following convention

*outerClassName$InnerClassName*

To facilitate availability of private static data members of outer class in inner class compiler makes following modifications:-

1. For each private static data member of outer class which is referenced in inner class a static accessor method is created in the outer class.
2. Reference of private static data member of outer class are replaced by call to accessor method in inner class.
3. NON-STATIC INNER class

A class that is defined within the scope of another class and is not qualified by **static** keyword is called **NON-STATIC INNER** class.

*It has following characterstics:-*

1. It cannot be instantiated independently, i.e. an object of outer class is required for the instantiation of non-static inner class.
2. It can access static as well as non-static data members of its outer class irrespective of their scope.

*Example Program for non-static inner class –*

**package** InnerClassOuterClass;

**class** A

{

**private** **static** **int** *a*;

**private** **int** b;

**public** A(**int** x)

{

b=x;

}

**static**

{

*a*=5;

System.*out*.println("A is loaded...");

}

**public** **void** display()

{

System.*out*.println("a of this class is : " + *a*);

System.*out*.println("b of this class is : " + b);

}

// non-static inner class starts

**public** **class** B

{

**int** c;

**public** B(**int** x)

{

c=x;

}

**public** **void** display()

{

System.*out*.println("a of outer class is : " + *a*);

System.*out*.println("b of this object is : " + b);

System.*out*.println("c of this object is : " + c);

}

}

// non-static inner class ends.

}

**class** InnerTest

{

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

{

System.*out*.println("Initiating outer class..");

A x = **new** A(10);

System.*out*.println("Initiating non-static inner class..");

A.B y = x.**new** B(20);

System.*out*.println("invoking display() method on outer class object");

x.display();

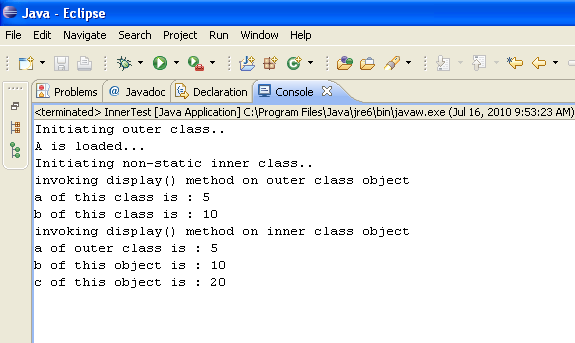
System.*out*.println("invoking display() method on inner class object");

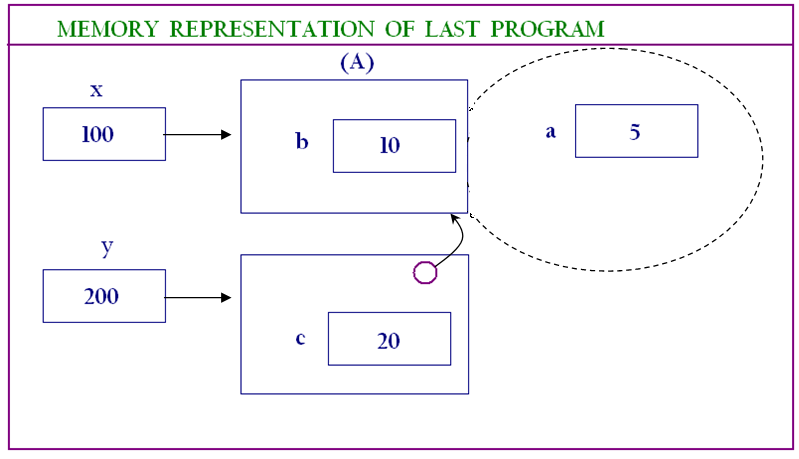
y.display();

}

}

Output –





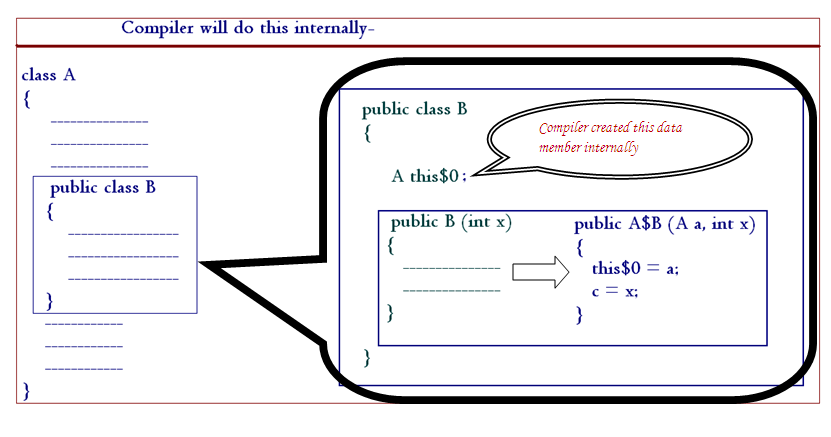
**Syntax of creating object of non-static inner class outside the scope of outer class:-**

*OuterClassName outerRef = new OuterClassName(-);*

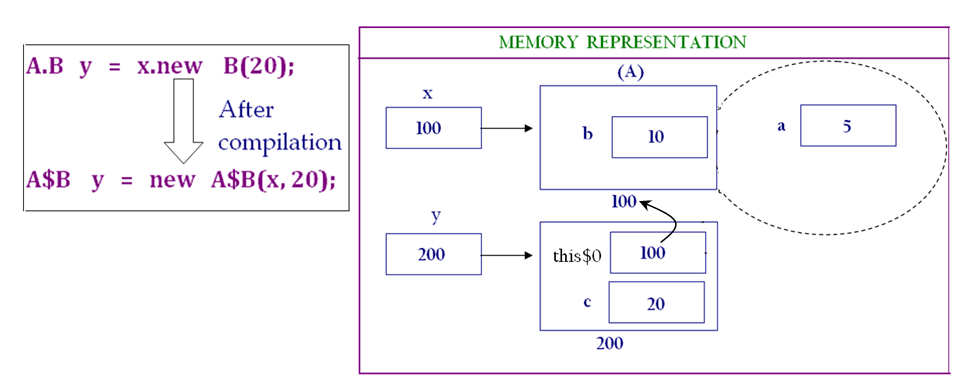
*OuterClassName. InnerClassName innerRef = outerRef.new InnerClassName(-);*

To facilitate availability of non-static data members of outer class in the non-static inner class compiler makes following modifications to the outer and inner classes:

1. **In the inner class :** 
   1. A data member of type outer class is contained in the inner class.

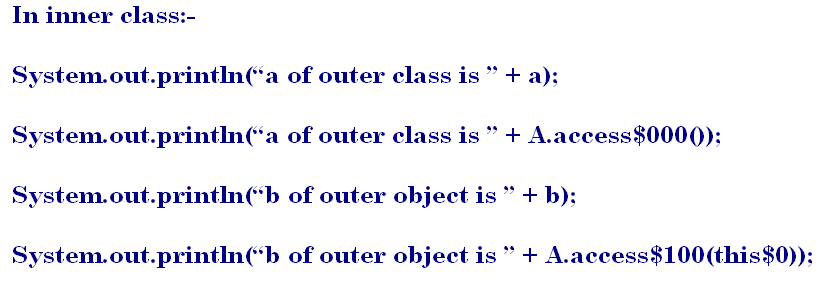
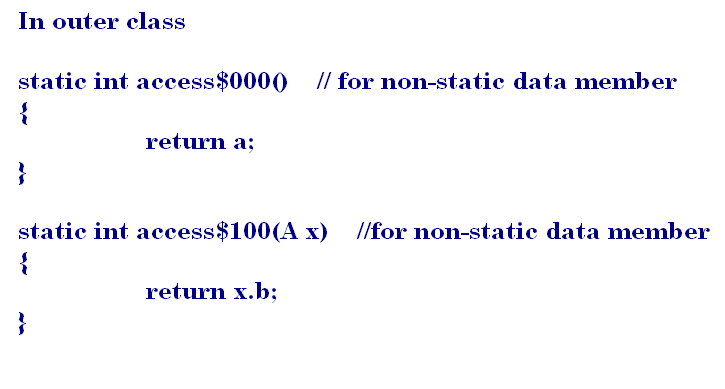


* 1. Constructor of inner class is modified to receive an extra parameter of type outer class.



1. **In the Outer class:**

For each private static data member of outer class static accessor method is created in the outer class.



**INNER CLASSES** provide a convenient mechanism of implementing **has-a** relation between classes. Non-static inner classes represents an intimate relation between outer & inner class because

1. Inner class is a member of outer class which means that outer class is incomplete without inner class.
2. Inner class is defined to provide a service to the outer class i.e. existence of inner class depends on the existence of outer class.

-----------------------------------------------------------X--------------------------------------------------------

**Date: 04.07.10**

**PACKAGE**

A package is a logical container that contains logically related classes, interfaces & subpackages.

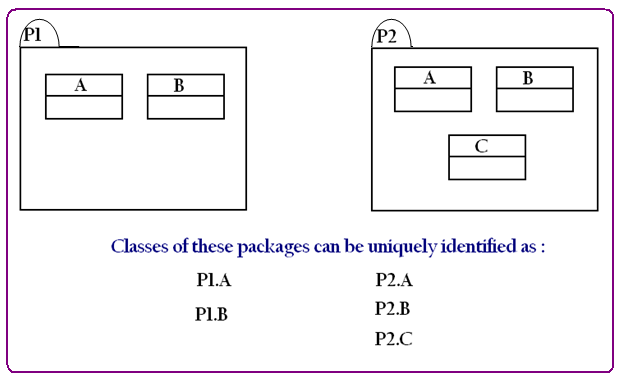
Packages are used to provide unique namespace to classes and packages provide a mechanism of enforcing scope.

Three parties are always involved in programming:-

1. Technology Provider
2. Application Developer
3. Third Party Vendor

Packages provide a mechanism of uniquely identifying classes by associating package name to the class name i.e. once a class is associated to a package it is referred using the fully qualified name.

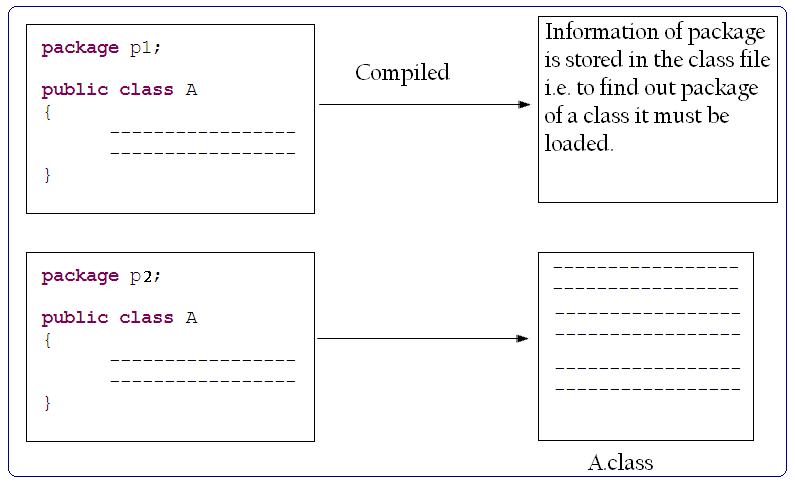
packageName.className



In order to associate a class to a package, **package** keyword is used as the first statement in a class definition.

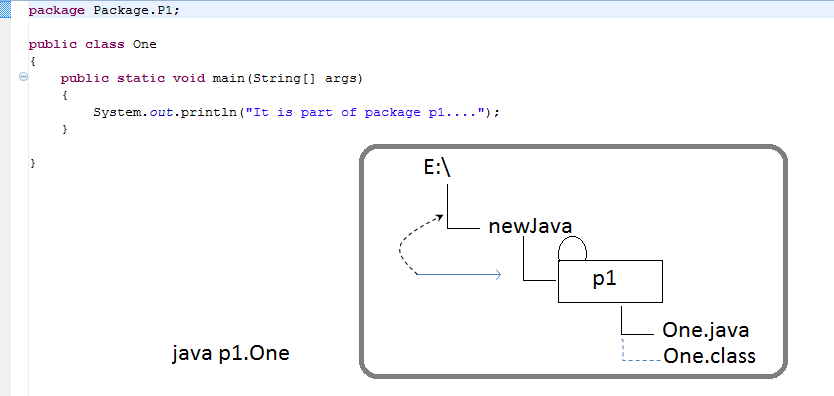
Syntax:-

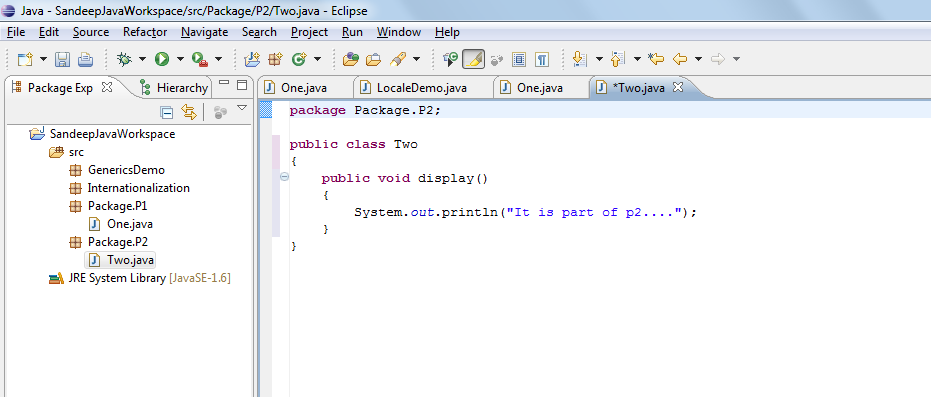
package pkgName;



To provide physical separation of classes of different packages classes of a package are saved in a folder of the same name as the package.

**NOTE**: Concept of package is applied only on classes not on java files.





package p2;

**public class** Two

{

**public void** display()

{

System.*out*.println("It is part of p2....");

}

}

**package** p1;

**public** **class** One

{

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

{

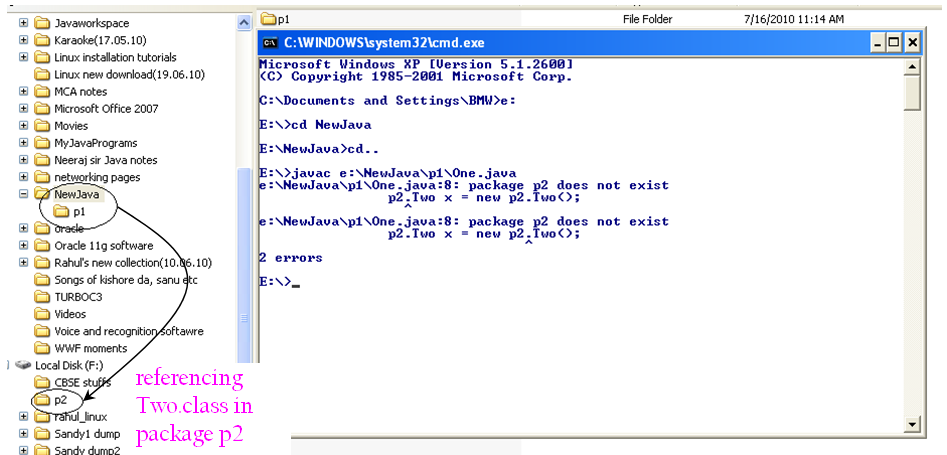
System.*out*.println("Referencing Two of p2....");

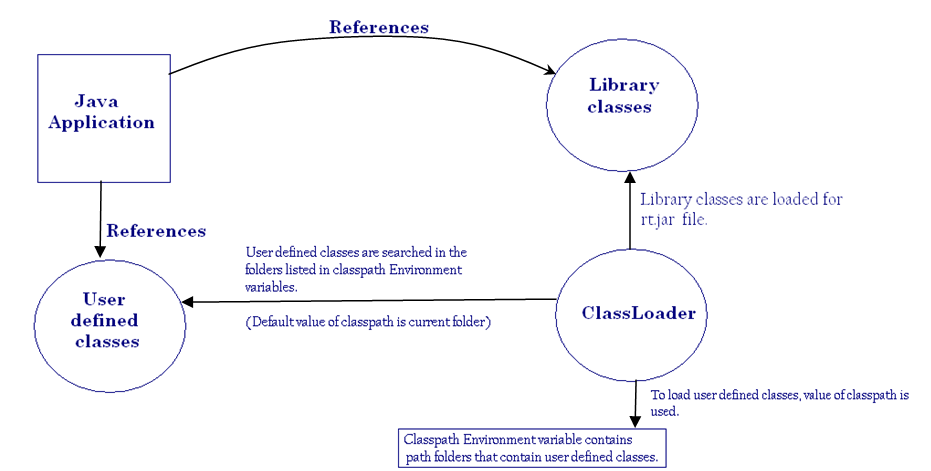
p2.Two x = **new** p2.Two();

x.display();

}

}





Import keyword is a convenience provided to java developers by the java compiler to refer classes of one package into another without using their fully qualified name.

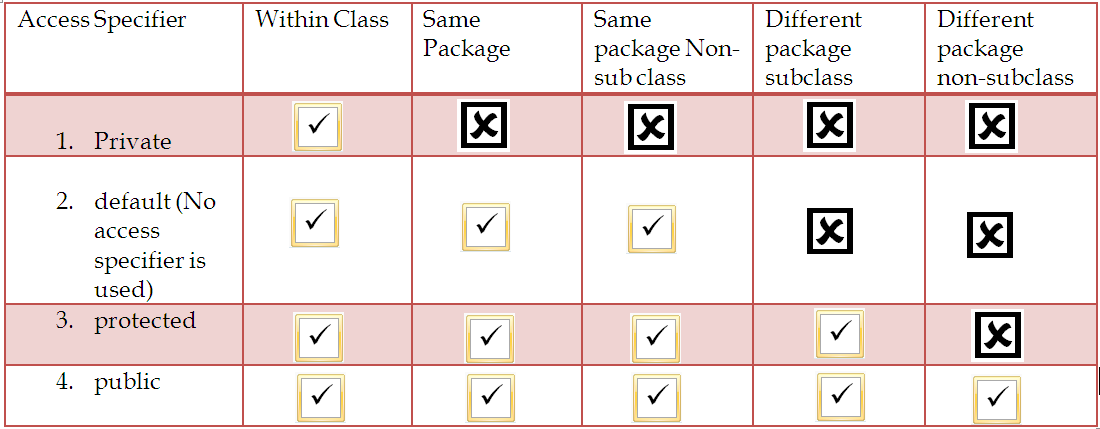
Syntax –

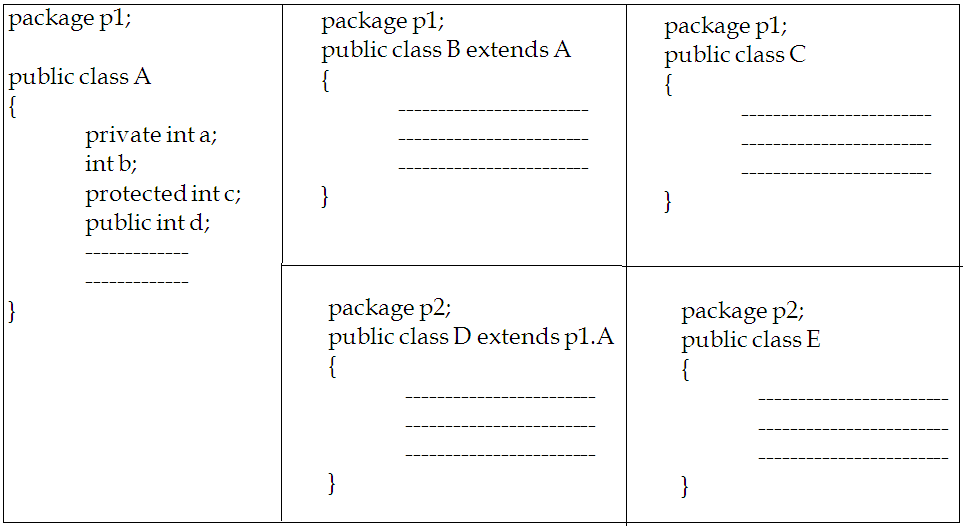
*import pkgName.className;*

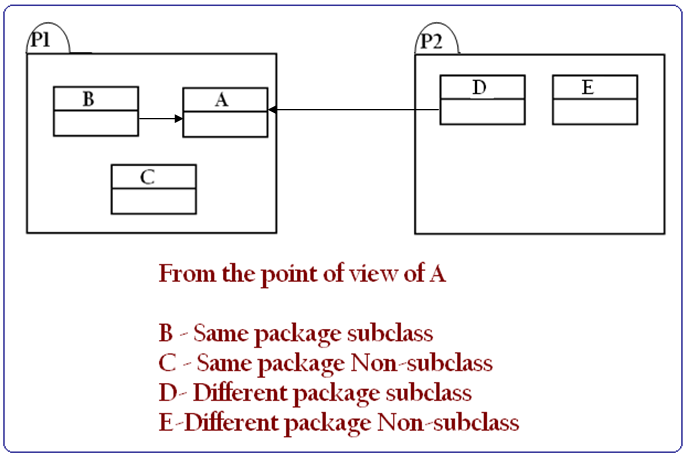
**or**

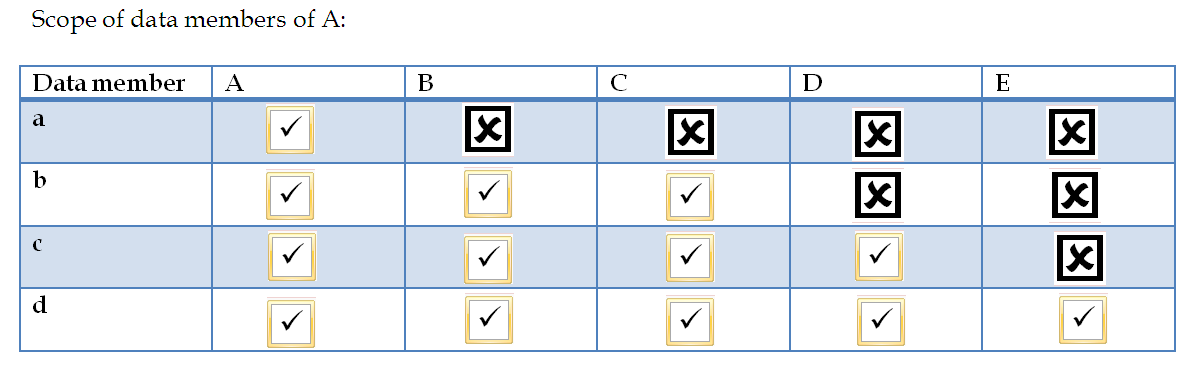
*import pkgName.\*;*

**Dated : 10.07.10**









package p1;

public class A

{

protected void display()

{

System.out.println(“protected method of A invoked.”);

}

}

package p2;

public class B extends A

{

public void show()

{

System.out.println(“Show of subclass invoked.”);

}

}

package p2;

public class C

{

public static void main(String args[])

{

B x = new B();

x.show();

x.display(); //Compilation error

}

}

package p2;

public class B extends p1.A

{

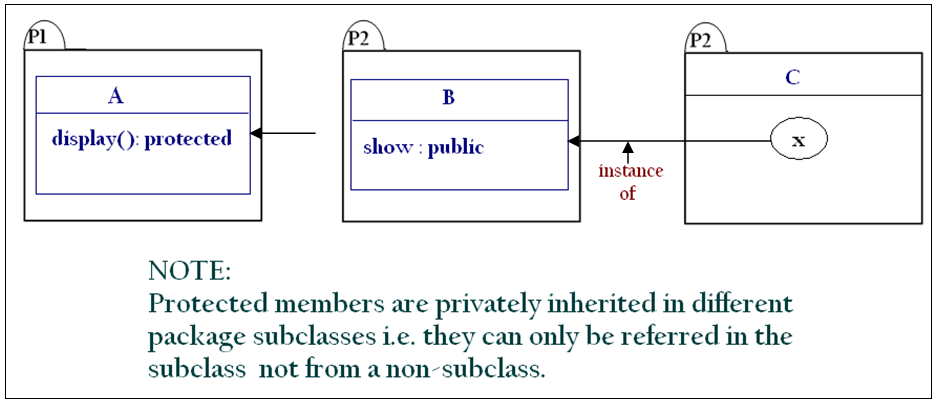
public void show()

{

System.out.println(“Show of subclass invoked.”);

}

}



----------------------------------------------X-----------------------------------------------

Now,

package p2;

public class B extends p1.A

{

public void show()

{

System.out.println(“Show of subclass invoked.”);

}

public static void main(String args[])

{

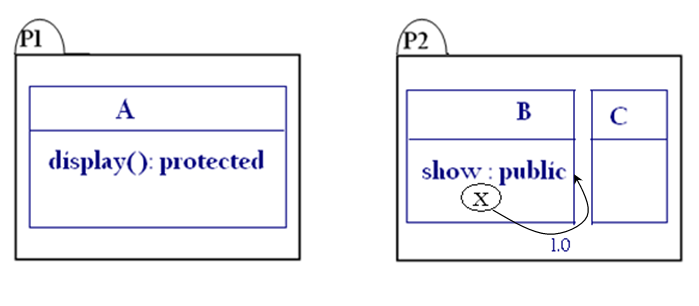
B x = new B();

x.show();

x.display(); //Compilation error

}

}



----------------------------------------------X-----------------------------------------------

package p2;

public class B extends p1.A

{

public void show()

{

System.out.println(“Show of subclass invoked.”);

}

public void display()

{

System.out.println(“display of subclass invoked.”);

}

public static void main(String args[])

{

B x = new B();

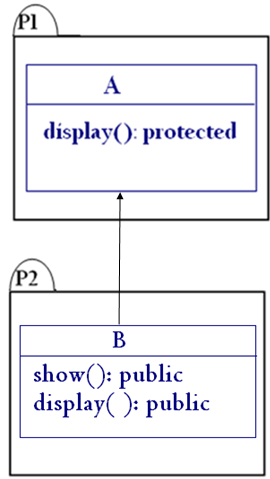
p1.A y = x;

x.show();

x.display(); //Shall not Compile

}

}



Protected members of a class can only be referred from a different package using the reference variables of those classes which are defined in the invoking package.

----------------------------------------------X-----------------------------------------------