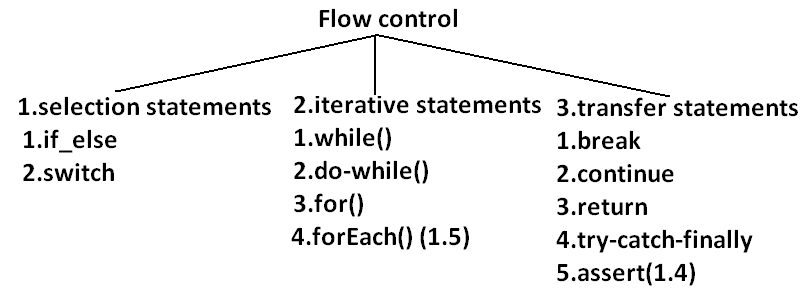
**Flow Control**

* Flow control describes the order in which all the statements will be executed at run time.

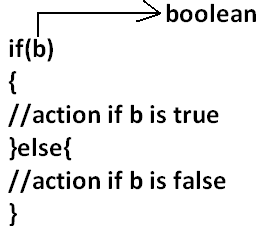
**Diagram:**



**Selection statements:**

1. **if-else:**

**syntax:**



* The argument to the if statement should be Boolean if we are providing any other type we will get “compile time error”.

**EXAMPLE 1:**

public class ExampleIf{

public static void main(String args[]){

int x=0;

if(x)

{

System.out.println("hello");

}else{

System.out.println("hi");

}}}

**OUTPUT:**

Compile time error:

D:\Java>javac ExampleIf.java

ExampleIf.java:4: incompatible types

found : int

required: boolean

if(x)

**EXAMPLE 2:**

public class ExampleIf{

public static void main(String args[]){

int x=10;

if(x=20)

{

System.out.println("hello");

}else{

System.out.println("hi");

}}}

**OUTPUT:**

Compile time error

D:\Java>javac ExampleIf.java

ExampleIf.java:4: incompatible types

found : int

required: boolean

if(x=20)

**EXAMPLE 3:**

public class ExampleIf{

public static void main(String args[]){

int x=10;

if(x==20)

{

System.out.println("hello");

}else{

System.out.println("hi");

}}}

**OUTPUT:**

Hi

**EXAMPLE 4:**

public class ExampleIf{

public static void main(String args[]){

boolean b=false;

if(b=true)

{

System.out.println("hello");

}else{

System.out.println("hi");

}}}

**OUTPUT:**

Hello

**EXAMPLE 5:**

public class ExampleIf{

public static void main(String args[]){

boolean b=false;

if(b==true)

{

System.out.println("hello");

}else{

System.out.println("hi");

}}}

**OUTPUT:**

Hi

* Both **else** and **curly braces** are optional.
* Without curly braces we can take only one statement under if, but it should not be declarative statement.

**EXAMPLE 6:**

public class ExampleIf{

public static void main(String args[]){

if(true)

System.out.println("hello");

}}

**OUTPUT:**

Hello

**EXAMPLE 7:**

public class ExampleIf{

public static void main(String args[]){

if(true);

}}

**OUTPUT:**

No output

**EXAMPLE 8:**

public class ExampleIf{

public static void main(String args[]){

if(true)

int x=10;

}}

**OUTPUT:**

Compile time error

D:\Java>javac ExampleIf.java

ExampleIf.java:4: '.class' expected

int x=10;

ExampleIf.java:4: not a statement

int x=10;

**EXAMPLE 9:**

public class ExampleIf{

public static void main(String args[]){

if(true){

int x=10;

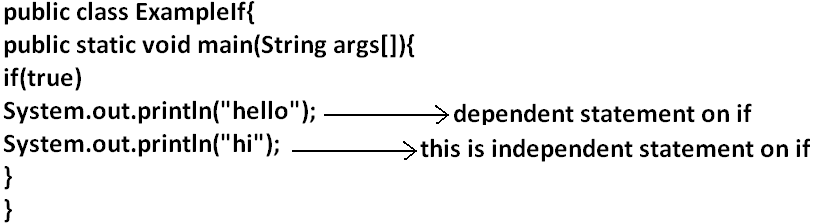
}}}

**OUTPUT:**

D:\Java>javac ExampleIf.java

D:\Java>java ExampleIf

**EXAMPLE 10:**



**OUTPUT:**

Hello

Hi

* Semicolon is a valid java statement which is called empty statement and it won’t produce any output.
* If several options are available then it is not recommended to use **if-else** we should go for **switch statement**.

**Switch:**

**Syntax:**

switch(x)

{

case 1:

action1

case 2:

action2

.

.

.

default:

default action

}

* **Curly braces are mandatory**.
* Both **case** and **default** are optional.
* Every statement inside switch must be under some case (or) default. Independent statements are not allowed.

**EXAMPLE 1:**

public class ExampleSwitch{

public static void main(String args[]){

switch(x)

{

System.out.println("hello");

}}}

**OUTPUT:**

Compile time error.

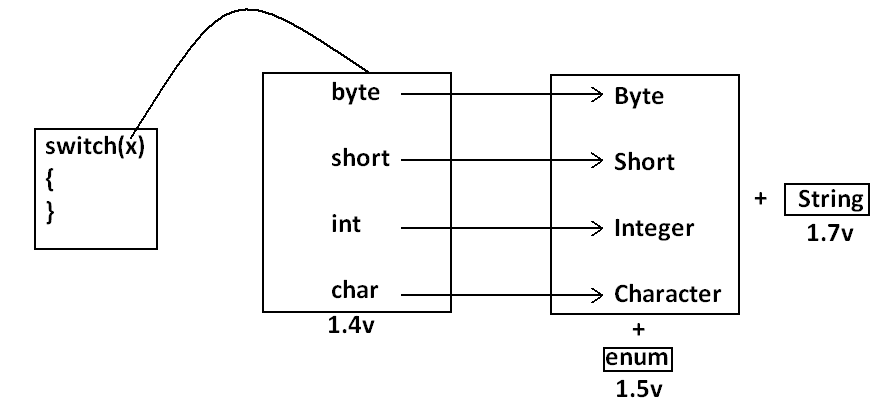
D:\Java>javac ExampleSwitch.java

ExampleSwitch.java:5: case, default, or '}' expected

System.out.println("hello");

* Until 1.4 version the allow types for the switch argument are byte, short, char, int but from 1.5 version on wards the corresponding wrapper classes (Byte, Short, Character, Integer) and “**enum**” types are allowed.

**DIAGRAM:**



* Every case label should be “compile time constant” otherwise we will get compile time error.

**EXAMPLE 2:**

public class ExampleSwitch{

public static void main(String args[]){

int x=10;

int y=20;

switch(x)

{

case 10:

System.out.println("10");

case y:

System.out.println("20");

}}}

**OUTPUT:**

Compile time error

D:\Java>javac ExampleSwitch.java

ExampleSwitch.java:9: constant expression required

case y:

* If we declare y as final we won’t get any compile time error.

**EXAMPLE 3:**

public class ExampleSwitch{

public static void main(String args[]){

int x=10;

final int y=20;

switch(x)

{

case 10:

System.out.println("10");

case y:

System.out.println("20");

}}}

**OUTPUT:**

10

20

* Switch argument and case label can be expressions also, **but case should be constant expression.**

**EXAMPLE 4:**

public class ExampleSwitch{

public static void main(String args[]){

int x=10;

switch(x+1)

{

case 10:

case 10+20:

case 10+20+30:

}}}

**OUTPUT:**

No output.

* Every case label should be within the range of switch argument type.

**EXAMPLE 5:**

public class ExampleSwitch{

public static void main(String args[]){

byte b=10;

switch(b)

{

case 10:

System.out.println("10");

case 100:

System.out.println("100");

case 1000:

System.out.println("1000");

}}}

**OUTPUT:**

Compile time error

D:\Java>javac ExampleSwitch.java

ExampleSwitch.java:10: possible loss of precision

found : int

required: byte

case 1000:

* Duplicate case labels are not allowed.

**EXAMPLE 6:**

public class ExampleSwitch{

public static void main(String args[]){

int x=10;

switch(x)

{

case 97:

System.out.println("97");

case 99:

System.out.println("99");

case 'a':

System.out.println("100");

}}}

**OUTPUT:**

Compile time error.

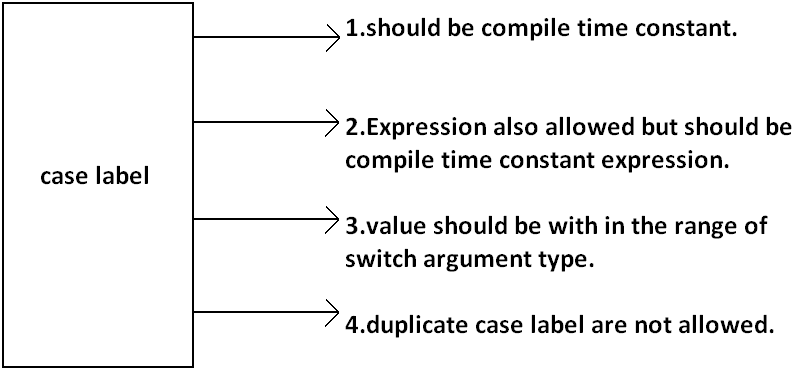
D:\Java>javac ExampleSwitch.java

ExampleSwitch.java:10: duplicate case label

case 'a':

**CASE SUMMARY:**

**DIAGRAM:**

****

**FALL-THROUGH INSIDE THE SWITCH:**

* With in the switch statement if any case is matched from that case onwards all statements will be executed until end of the switch (or) break. This is call “fall-through” inside the switch .

**EXAMPLE 7:**

public class ExampleSwitch{

public static void main(String args[]){

int x=0;

switch(x)

{

case 0:

System.out.println("0");

case 1:

System.out.println("1");

break;

case 2:

System.out.println("2");

default:

System.out.println("default");

}}}

**OUTPUT:**

**X=0** **x=1** **x=2** **x=3**

0 1 2 default

1 default

**DEFAULT CASE:**

* With in the switch we can take the **default** any where, but at most once it is convension to take default as last case.

**EXAMPLE 8:**

public class ExampleSwitch{

public static void main(String args[]){

int x=0;

switch(x)

{

default:

System.out.println("default");

case 0:

System.out.println("0");

break;

case 1:

System.out.println("1");

case 2:

System.out.println("2");

}}}

**OUTPUT:**

**X=0**  **x=1** **x=2** **x=3**

0 1 2 default

2 0

**ITERATIVE STATEMENTS:**

**While loop**: if we don’t know the no of iterations in advance then best loop is while loop:

**EXAMPLE 1:**

while(rs.next())

{

}

**EXAMPLE 2:**

while(e.hasMoreelEments())

{

----------

----------

----------

}

**EXAMPLE 3:**

while(itr.hasNext())

{

----------

----------

----------

}

* The argument to the while statement should be Boolean type. If we are using any other type we will get compile time error.

**EXAMPLE 1:**

public class ExampleWhile{

public static void main(String args[]){

while(1)

{

System.out.println("hello");

}}}

**OUTPUT:**

Compile time error.

D:\Java>javac ExampleWhile.java

ExampleWhile.java:3: incompatible types

found : int

required: boolean

while(1)

* Curly braces are optional and without curly braces we can take only one statement which should not be declarative statement.

**EXAMPLE 2:**

public class ExampleWhile{

public static void main(String args[]){

while(true)

System.out.println("hello");

}}

**OUTPUT:**

Hello (infinite times).

**EXAMPLE 3:**

public class ExampleWhile{

public static void main(String args[]){

while(true);

}}

**OUTPUT:**

No output.

**EXAMPLE 4:**

public class ExampleWhile{

public static void main(String args[]){

while(true)

int x=10;

}}

**OUTPUT:**

Compile time error.

D:\Java>javac ExampleWhile.java

ExampleWhile.java:4: '.class' expected

int x=10;

ExampleWhile.java:4: not a statement

int x=10;

**EXAMPLE 5:**

public class ExampleWhile{

public static void main(String args[]){

while(true)

{

int x=10;

}}}

**OUTPUT:**

No output.

Unreachable statement in while:

**EXAMPLE 6:**

public class ExampleWhile{

public static void main(String args[]){

while(true)

{

System.out.println("hello");

}

System.out.println("hi");

}}

**OUTPUT:**

Compile time error.

D:\Java>javac ExampleWhile.java

ExampleWhile.java:7: unreachable statement

System.out.println("hi");

**EXAMPLE 7:**

public class ExampleWhile{

public static void main(String args[]){

while(false)

{

System.out.println("hello");

}

System.out.println("hi");

}}

**OUTPUT:**

D:\Java>javac ExampleWhile.java

ExampleWhile.java:4: unreachable statement

{

**EXAMPLE 8:**

public class ExampleWhile{

public static void main(String args[]){

int a=10,b=20;

while(a<b)

{

System.out.println("hello");

}

System.out.println("hi");

}}

**OUTPUT:**

Hello (infinite times).

**EXAMPLE 9:**

public class ExampleWhile{

public static void main(String args[]){

final int a=10,b=20;

while(a<b)

{

System.out.println("hello");

}

System.out.println("hi");

}}

**OUTPUT:**

Compile time error.

D:\Java>javac ExampleWhile.java

ExampleWhile.java:8: unreachable statement

System.out.println("hi");

**EXAMPLE 10:**

public class ExampleWhile{

public static void main(String args[]){

final int a=10;

while(a<20)

{

System.out.println("hello");

}

System.out.println("hi");

}}

**OUTPUT:**

D:\Java>javac ExampleWhile.java

ExampleWhile.java:8: unreachable statement

System.out.println("hi");

Note:

* Every final variable will be replaced with the corresponding value by compiler.
* If any operation involves only constants then compiler is responsible to perform that operation.
* If any operation involves at least one variable compiler won’t perform that operation. At runtime jvm is responsible to perform that operation.

**EXAMPLE 11:**

public class ExampleWhile{

public static void main(String args[]){

int a=10;

while(a<20)

{

System.out.println("hello");

}

System.out.println("hi");

}}

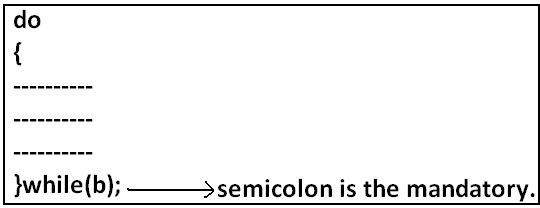
**OUTPUT:**

Hello (infinite times).

**Do-while:**

* If we want to execute loop body at least once then we should go for do-while.

**Syntax:**



* Curly braces are optional.
* Without curly braces we can take only one statement between do and while and it should not be declarative statement.

**Example 1:**

public class ExampleDoWhile{

public static void main(String args[]){

do

System.out.println("hello");

while(true);

}}

**Output:**

Hello (infinite times).

**Example 2:**

public class ExampleDoWhile{

public static void main(String args[]){

do;

while(true);

}}

**Output:**

Compile successful.

**Example 3:**

public class ExampleDoWhile{

public static void main(String args[]){

do

int x=10;

while(true);

}}

**Output:**

D:\Java>javac ExampleDoWhile.java

ExampleDoWhile.java:4: '.class' expected

int x=10;

ExampleDoWhile.java:4: not a statement

int x=10;

ExampleDoWhile.java:4: ')' expected

int x=10;

**Example 4:**

public class ExampleDoWhile{

public static void main(String args[]){

do

{

int x=10;

}while(true);

}}

**Output:**

Compile successful.

**Example 5:**

public class ExampleDoWhile{

public static void main(String args[]){

do while(true)

System.out.println("hello");

while(true);

}}

**Output:**

Hello (infinite times).

**Rearrange the above example:**

public class ExampleDoWhile{

public static void main(String args[]){

do

while(true)

System.out.println("hello");

while(true);

}}

**Output:**

Hello (infinite times).

**Example 6:**

public class ExampleDoWhile{

public static void main(String args[]){

do

while(true);

}}

**Output:**

Compile time error.

D:\Java>javac ExampleDoWhile.java

ExampleDoWhile.java:4: while expected

while(true);

ExampleDoWhile.java:5: illegal start of expression

}

**Unreachable statement in do while:**

**Example 7:**

public class ExampleDoWhile{

public static void main(String args[]){

do

{

System.out.println("hello");

}

while(true);

System.out.println("hi");

}}

**Output:**

Compile time error.

D:\Java>javac ExampleDoWhile.java

ExampleDoWhile.java:8: unreachable statement

System.out.println("hi");

**Example 8:**

public class ExampleDoWhile{

public static void main(String args[]){

do

{

System.out.println("hello");

}

while(false);

System.out.println("hi");

}}

**Output:**

Hello

Hi

**Example 9:**

public class ExampleDoWhile{

public static void main(String args[]){

int a=10,b=20;

do

{

System.out.println("hello");

}

while(a<b);

System.out.println("hi");

}}

**Output:**

Hello (infinite times).

**Example 10:**

public class ExampleDoWhile{

public static void main(String args[]){

int a=10,b=20;

do

{

System.out.println("hello");

}

while(a>b);

System.out.println("hi");

}}

**Output:**

Hello

Hi

**Example 11:**

public class ExampleDoWhile{

public static void main(String args[]){

final int a=10,b=20;

do

{

System.out.println("hello");

}

while(a<b);

System.out.println("hi");

}}

**Output:**

Compile time error.

D:\Java>javac ExampleDoWhile.java

ExampleDoWhile.java:9: unreachable statement

System.out.println("hi");

**Example 12:**

public class ExampleDoWhile{

public static void main(String args[]){

final int a=10,b=20;

do

{

System.out.println("hello");

}

while(a>b);

System.out.println("hi");

}}

**Output:**

D:\Java>javac ExampleDoWhile.java

D:\Java>java ExampleDoWhile

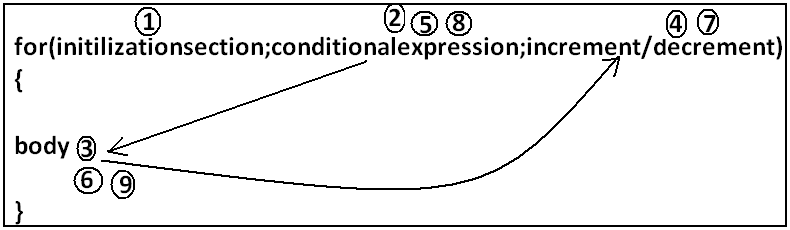
Hello

Hi

**For Loop:**

* This is the most commonly used loop and best suitable if we know the no of iterations in advance.

**Syntax:**



1. **Initilizationsection:**

* This section will be executed only once.
* Here usually we can declare loop variables and we will perform initialization.
* We can declare multiple variables but should be of the same type and we can’t declare different type of variables.

**Example:**

1. Int i=0,j=0; **valid**
2. Int i=0,Boolean b=true; **invalid**
3. Int i=0,int j=0; **invalid**

* In initialization section we can take any valid java statement including “s.o.p” also.

**Example 1:**

public class ExampleFor{

public static void main(String args[]){

int i=0;

for(System.out.println("hello u r sleeping");i<3;i++){

System.out.println("no boss, u only sleeping");

}}}

**Output:**

D:\Java>javac ExampleFor.java

D:\Java>java ExampleFor

Hello u r sleeping

No boss, u only sleeping

No boss, u only sleeping

No boss, u only sleeping

1. **Conditional check:**

* We can take any java expression but should be of the type Boolean.
* Conditional expression is optional and if we are not taking any expression compiler will place true.

1. **Increment and decrement section:**

* Here we can take any java statement including s.o.p also.

**Example:**

public class ExampleFor{

public static void main(String args[]){

int i=0;

for(System.out.println("hello");i<3;System.out.println("hi")){

i++;

}}}

**Output:**

D:\Java>javac ExampleFor.java

D:\Java>java ExampleFor

Hello

Hi

Hi

Hi

* All 3 parts of for loop are independent of each other and all optional.

**Example:**

public class ExampleFor{

public static void main(String args[]){

for(;;){

System.out.println("hello");

}}}

**Output:**

Hello (infinite times).

* Curly braces are optional and without curly braces we can take exactly one statement and it should not be declarative statement.

**Unreachable statement in for loop:**

**Example 1:**

public class ExampleFor{

public static void main(String args[]){

for(int i=0;true;i++){

System.out.println("hello");

}

System.out.println("hi");

}}

**Output:**

Compile time error.

D:\Java>javac ExampleFor.java

ExampleFor.java:6: unreachable statement

System.out.println("hi");

**Example 2:**

public class ExampleFor{

public static void main(String args[]){

for(int i=0;false;i++){

System.out.println("hello");

}

System.out.println("hi");

}}

**Output:**

Compile time error.

D:\Java>javac ExampleFor.java

ExampleFor.java:3: unreachable statement

for(int i=0;false;i++){

**Example 3:**

public class ExampleFor{

public static void main(String args[]){

for(int i=0;;i++){

System.out.println("hello");

}

System.out.println("hi");

}}

**Output:**

Compile time error.

D:\Java>javac ExampleFor.java

ExampleFor.java:6: unreachable statement

System.out.println("hi");

**Example 4:**

public class ExampleFor{

public static void main(String args[]){

int a=10,b=20;

for(int i=0;a<b;i++){

System.out.println("hello");

}

System.out.println("hi");

}}

**Output:**

Hello (infinite times).

**Example 5:**

public class ExampleFor{

public static void main(String args[]){

final int a=10,b=20;

for(int i=0;a<b;i++){

System.out.println("hello");

}

System.out.println("hi");

}}

**Output:**

D:\Java>javac ExampleFor.java

ExampleFor.java:7: unreachable statement

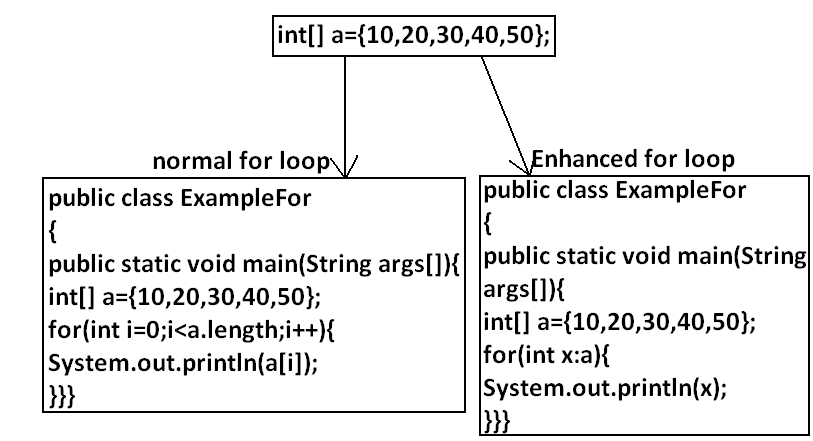
System.out.println("hi");

**For each:**

* For each Introduced in 1.5version.
* **Best suitable to retrieve the elements of arrays and collections**.

**Example 1:** Write code to print the elements of single dimensional array by normal for loop and enhanced for loop.

**Example:**



**Output:**

D:\Java>javac ExampleFor.java

D:\Java>java ExampleFor

10

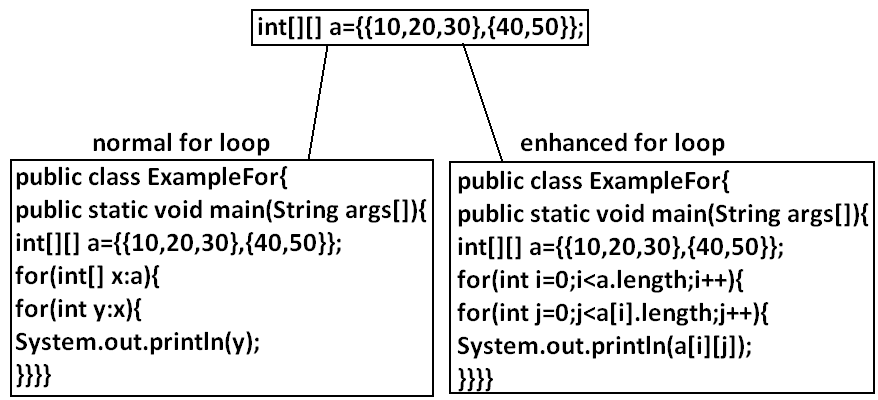
20

30

40

50

**Example 2:** Write code to print the elements of 2 dimensional arrays by using normal for loop and enhanced for loop.



**Example 3:** Write equivalent code by For Each loop for the following for loop.

public class ExampleFor{

public static void main(String args[]){

for(int i=0;i<10;i++)

{

System.out.println("hello");

}}}

**Output:**

D:\Java>javac ExampleFor1.java

D:\Java>java ExampleFor1

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

Hello

* We can’t write equivalent for each loop.
* For each loop is the more convenient loop to retrieve the elements of arrays and collections, but its main limitation is it is not a general purpose loop.

**Transfer statements:**

**Break statement:**

* We can use break statement in the following cases.

1. Inside switch to stop fall-through.
2. Inside loops to break the loop based on some condition.
3. Inside label blocks to break block execution based on some condition.

**Example 1:**

class Test{

public static void main(String args[]){

int x=10;

l1:

{

System.out.println("hello");

if(x==10)

break l1;

System.out.println("hi");

}

System.out.println("end");

}}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

Hello

End

* These are the only places where we can use break statement. If we are using anywhere else we will get compile time error.

**Example:**

class Test{

public static void main(String args[]){

int x=10;

if(x==10)

break;

System.out.println("hello");

}}

**Output:**

Compile time error.

D:\Java>javac Test.java

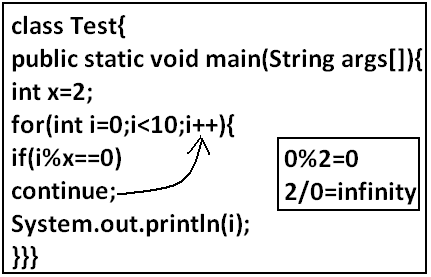
Test.java:5: break outside switch or loop

break;

**Continue statement:**

* We can use continue statement to skip current iteration and continue for the next iteration.

**Example:**



**Output:**

D:\Java>javac Test.java

D:\Java>java Test

1

3

5

7

9

* We can use continue only inside loops if we are using anywhere else we will get compile time error saying “continue outside of loop”.

**Example:**

class Test

{

public static void main(String args[]){

int x=10;

if(x==10);

continue;

System.out.println("hello");

}

}

**Output:**

Compile time error.

D:\Enum>javac Test.java

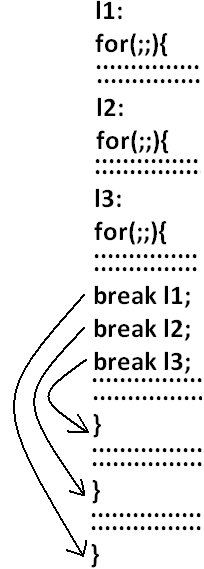
Test.java:6: continue outside of loop

continue;

**Labeled break and continue statements:**

* In the nested loops to break (or) continue a particular loop we should go for labeled break and continue statements.

**Syntax:**



**Example:**

class Test

{

public static void main(String args[]){

l1:

for(int i=0;i<3;i++)

{

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

{

if(i==j)

break;

System.out.println(i+"........."+j);

}}}}

**Break:**

1.........0

2.........0

2.........1

**Break l1:**

No output.

**Continue:**

0.........1

0.........2

1.........0

1.........2

2.........0

2.........1

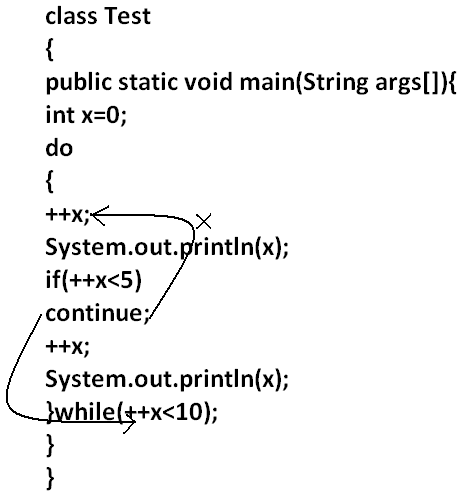
**Continue l1:**

1.........0

2.........0

2.........1

**Do-while vs continue (The most dangerous combination):**

****

**Output:**

1

4

6

8

10

* Compiler won’t check unreachability in the case of if-else it will check only in loops.

**Example 1:**

class Test

{

public static void main(String args[]){

while(true)

{

System.out.println("hello");

}

System.out.println("hi");

}

}

**Output:**

Compile time error.

D:\Enum>javac Test.java

Test.java:8: unreachable statement

System.out.println("hi");

**Example 2:**

class Test

{

public static void main(String args[]){

if(true)

{

System.out.println("hello");

}

else

{

System.out.println("hi");

}}}

**Output:**

Hello

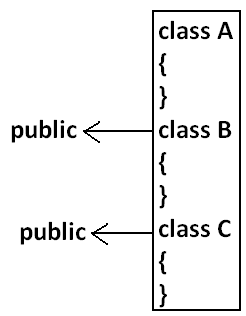
**Declaration and Access Modifiers**

1. Java source file structure
2. Class modifiers
3. Member modifiers
4. Interfaces

**Java source file structure:**

* A java program can contain any no. Of classes but at mot one class can be declared as public. “If there is a public class the name of the program and name of the public class must be matched otherwise we will get compile time error”.
* If there is no public class then any name we gives for java source file.

**Example:**



**Case1:**

* If there is no public class then we can use any name for java source file there are no restrictions.

**Example:**

A.java

B.java

C.java

Bhaskar.java

**case2:**

* If class B declared as public then the name of the program should be B.java otherwise we will get compile time error saying “class B is public, should be declared in a file named B.java”.

**Cae3:**

* If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying “class C is public, should be declared in a file named C.java”.
* It is highly recommended to take only one class for source file and name of the program (file) must be same as class name. This approach improves readability and understandability of the code.

**Example:**

class A

{

public static void main(String args[]){

System.out.println("A class main method is executed");

}

}

class B

{

public static void main(String args[]){

System.out.println("B class main method is executed");

}

}

class C

{

public static void main(String args[]){

System.out.println("C class main method is executed");

}

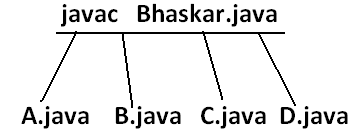
}

class D

{

}

**Output:**



D:\Java>java A

A class main method is executed

D:\Java>java B

B class main method is executed

D:\Java>java C

C class main method is executed

D:\Java>java D

Exception in thread "main" java.lang.NoSuchMethodError: main

D:\Java>java Bhaskar

Exception in thread "main" java.lang.NoClassDefFoundError: Bhaskar

* We can compile a java program but not java class in that program for every class one dot class file will be created.
* We can run a java class but not java source file whenever we are trying to run a class the corresponding class main method will be executed.
* If the class won’t contain main method then we will get runtime exception saying “NoSuchMethodError: main”.
* If we are trying to execute a java class and if the corresponding .class file is not available then we will get runtime execution saying “NoClassDefFoundError: Bhaskar”.

**Import statement:**

class Test{

public static void main(String args[]){

ArrayList l=new ArrayList();

}

}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:3: cannot find symbol

symbol : class ArrayList

location: class Test

ArrayList l=new ArrayList();

Test.java:3: cannot find symbol

symbol : class ArrayList

location: class Test

ArrayList l=new ArrayList();

* We can resolve this problem by using fully qualified name “java.util.ArrayList l=new java.util.ArrayList();”. But problem with using fully qualified name every time is it increases length of the code and reduces readability.
* We can resolve this problem by using import statements.

**Example:**

import java.util.ArrayList;

class Test{

public static void main(String args[]){

ArrayList l=new ArrayList();

}

}

**Output:**

D:\Java>javac Test.java

* Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly. This approach decreases length of the code and improves readability.

**Case 1: Types of Import Statements:**

* There are 2 types of import statements.

1. Explicit class import
2. Implicit class import.

**Explicit class import:**

**Example:** Import java.util.ArrayList

* This type of import is highly recommended to use because it improves readability of the code.
* Best suitable for Hi-Tech city where readability is important.

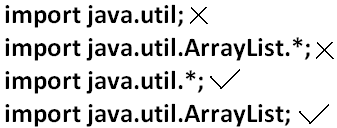
**Implicit class import:**

**Example**: import java.util.\*;

* It is never recommended to use because it reduces readability of the code.
* Bet suitable for Ameerpet where typing is important.

**Case2:**

**Which of the following import statements are valid?**



**Case3:**

* consider the following code.

class MyArrayList extends java.util.ArrayList

{

}

* The code compiles fine even though we are not using import statements because we used fully qualified name.
* Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not require to use fully qualified name.

**Case4:**

**Example:**

import java.util.\*;

import java.sql.\*;

class Test

{

public static void main(String args[])

{

Date d=new Date();

}}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:7: reference to Date is ambiguous, both class java.sql.Date in java.sql and class java.util.Date in java.util match

Date d=new Date();

**Note:** Even in the List case also we may get the same ambiguity problem because it is available in both UTIL and AWT packages.

**Case5:**

* While resolving class names compiler will always gives the importance in the following order.

1. Explicit class import
2. Classes present in current working directory.
3. Implicit class import.

**Example:**

import java.util.Date;

import java.sql.\*;

class Test

{

public static void main(String args[]){

Date d=new Date();

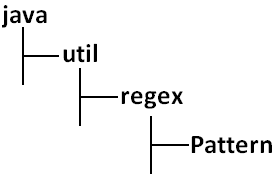
}}

* The code compiles fine and in this case util package Date will be considered.

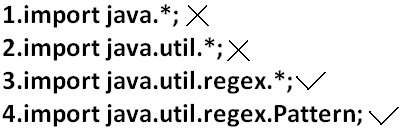
**Case6:**

* Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes.

**Example:**



**To use pattern class in our program directly which import statement is required?**



**Case7:**

* In any java program the following 2 packages are not require to import because these are available by default to every java program.

1. java.lang package
2. default package(current working directory)

**Case8:**

* “Import statement is totally compile time concept” if more no of imports are there then more will be the compile time but there is “no change in execution time”.

**Difference between C language #include and java language import.**

* In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.
* But in java import statement no “.class” will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding “.class” file will be loaded. Hence it follows “dynamic loading” or “load-on –demand” or “load-on-fly”.

**Static import:**

* This concept introduced in 1.5 versions. According to sun static import improves readability of the code but according to worldwide programming exports (like us) static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import.

**1.5 versions new features**

1. For-Each
2. Var-arg
3. Queue
4. Generics
5. Auto boxing and Auto unboxing
6. Co-varient return types
7. Annotations
8. Enum
9. Static import
10. String builder

* Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

**Without static import:**

class Test

{

public static void main(String args[]){

System.out.println(Math.sqrt(4));

System.out.println(Math.max(10,20));

System.out.println(Math.random());

}}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

2.0

20

0.841306154315576

**With static import:**

import static java.lang.Math.sqrt;

import static java.lang.Math.\*;

class Test

{

public static void main(String args[]){

System.out.println(sqrt(4));

System.out.println(max(10,20));

System.out.println(random());

}}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

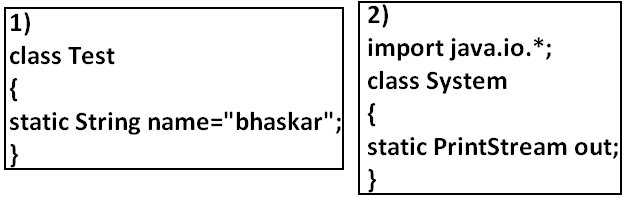
2.0

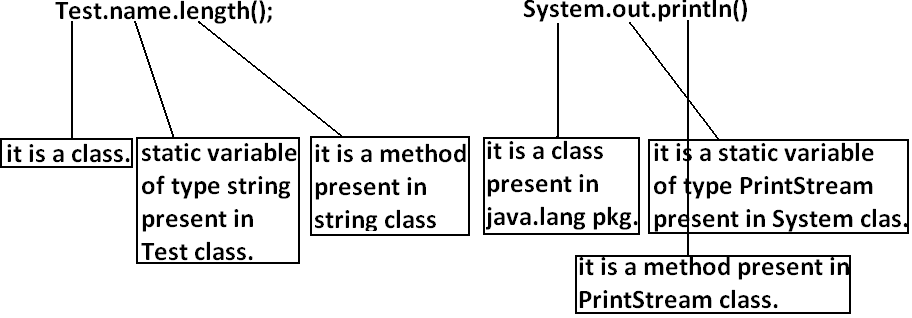
20

0.4302853847363891

**Explain about System.out.println statement?**

**Example 1 and example 2:**





**Example 3:**

import static java.lang.System.out;

class Test

{

public static void main(String args[]){

out.println("hello");

out.println("hi");

}}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

hello

hi

**Example 4:**

import static java.lang.Integer.\*;

import static java.lang.Byte.\*;

class Test

{

public static void main(String args[]){

System.out.println(MAX\_VALUE);

}}

**Output:**

Compile time error.

D:\Java>javac Test.java

Test.java:6: reference to MAX\_VALUE is ambiguous, both variable MAX\_VALUE in java.lang.Integer and variable MAX\_VALUE in java.lang.Byte match

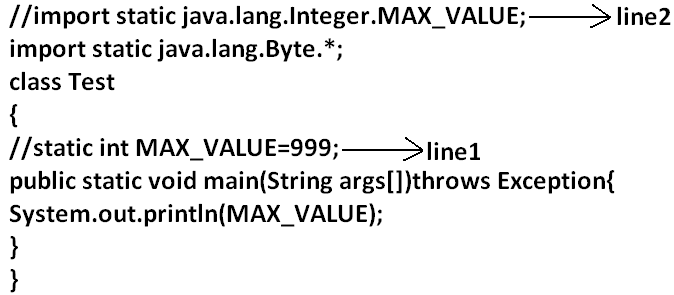
System.out.println(MAX\_VALUE);

**Note:** Two packages contain a class or interface with the same is very rare hence ambiguity problem is very rare in normal import.

* But 2 classes or interfaces can contain a method or variable with the same name is very common hence ambiguity problem is also very common in static import.
* While resolving static members compiler will give the precedence in the following order.

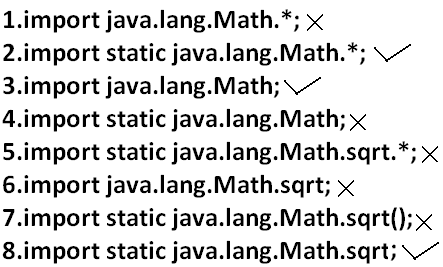
1. Current class static members
2. Explicit static import
3. implict static import.

**Example:**

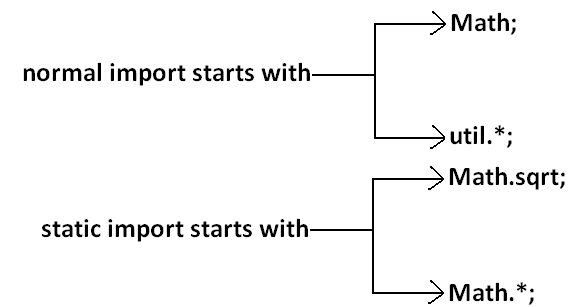


* If we comet line one then we will get Integer class MAX\_VALUE 2147483647.
* If we comet lines one and two then Byte class MAX\_VALUE will be considered 127.

**Which of the following import statements are valid?**



**Diagram:**



* Usage of static import reduces readability and creates confusion hence if there is no specific requirement never recommended to use static import.

**What is the difference between general import and static import?**

* We can use normal imports to import classes and interfaces of a package. whenever we are using normal import we can access class and interfaces directly by their short name it is not require to use fully qualified names.
* We can use static import to import static members of a particular class. whenever we are using static import it is not require to use class name we can access static members directly.

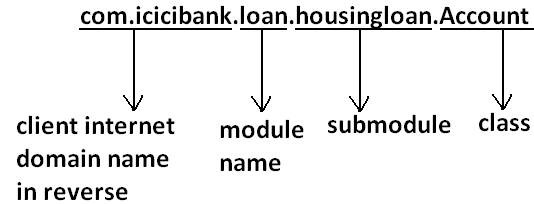
**Package statement:**

* It is an encapsulation mechanism to group related classes and interfaces into a single module.

**The main objectives of packages are:**

* To resolve name confects.
* To improve modularity of the application.
* To provide security.
* There is one universally accepted naming conversion for packages that is to use internet domain name in reverse.

**Example:**

****

**How to compile package program:**

**Example:**

package com.ksraojobs.itjobs;

class HydJobs

{

public static void main(String args[]){

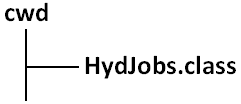
System.out.println("package demo");

}

}

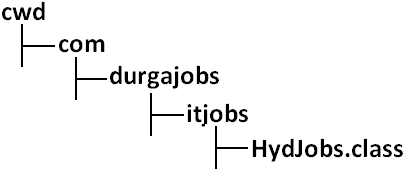
* Javac HydJobs.java generated class file will be placed in current working directory.

**Diagram:**



* Javac –d . HydJobs.java
* -d means destination to place generated class files “.” means current working directory.
* Generated class file will be placed into corresponding package structure.

**Diagram:**

****

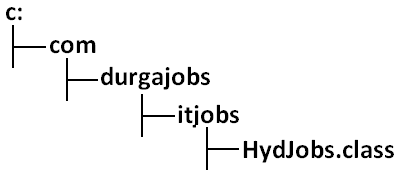
* If the specified package structure is not already available then this command itself will create the required package structure.
* As the destination we can use any valid directory.

If the specified destination is not available then we will get compile time error.

**Example:**

D:\Java>javac -d c: HydJobs.java

**Diagram:**



* If the specified destination is not available then we will get compile time error.

**Example:**

D:\Java>javac -d z: HydJobs.java

* If Z: is not available then we will get compile time error.

**How to execute package program:**

D:\Java>java com.ksraojobs.itjobs.HydJobs

* At the time of execution compulsory we should provide fully qualified name.

**Conclusion 1:**

* In any java program there should be at most one package statement that is if we are taking more than one package statement we will get compile time error.

**Example:**

package pack1;

package pack2;

class A

{

}

**Output:**

Compile time error.

D:\Java>javac A.java

A.java:2: class, interface, or enum expected

package pack2;

**Conclusion 2:**

* In any java program the 1st non cement statement should be package statement [if it is available] otherwise we will get compile time error.

**Example:**

import java.util.\*;

package pack1;

class A

{

}

**Output:**

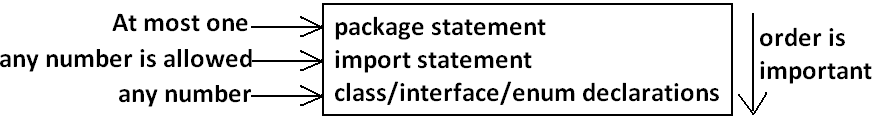
Compile time error.

D:\Java>javac A.java

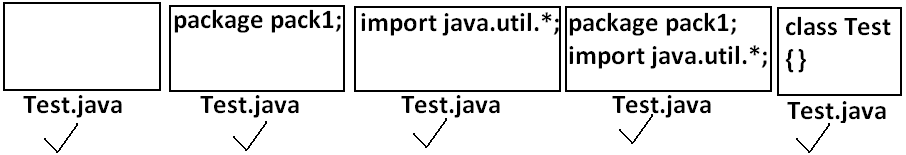
A.java:2: class, interface, or enum expected

package pack1;

**Java source file structure:**

****

* All the following are valid java programs.



**Note:** An empty source file is a valid java program.

**Class Modifiers**

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

1. Better this class can be accessible from anywhere or not.
2. Better 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.

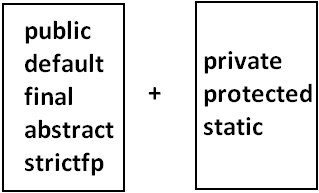
D:\Java>javac Test.java

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.

**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 . 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.

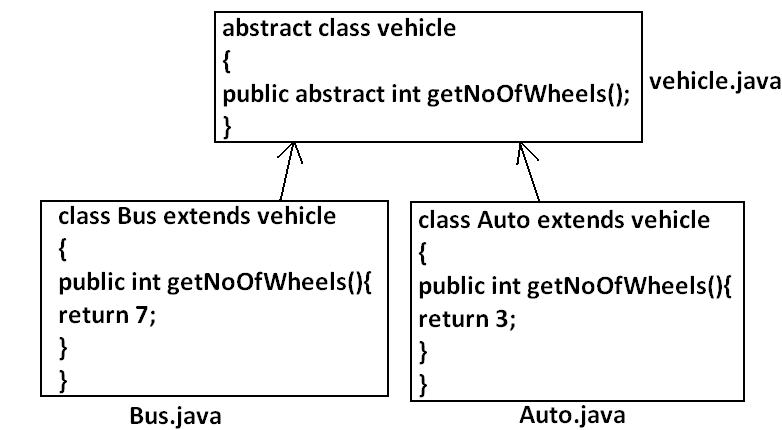
**EXAMPLE:**



* Child classes are responsible to provide implementation for parent class abstract methods.

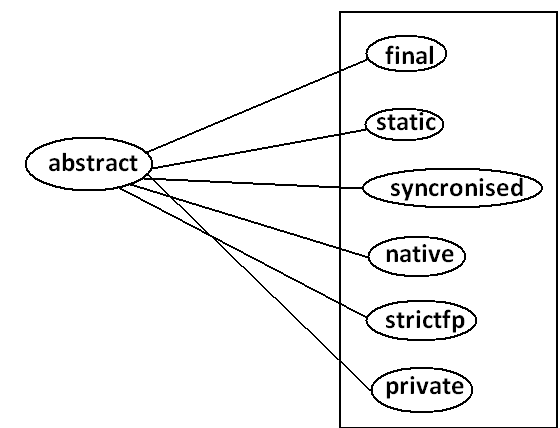
**EXAMPLE:**

**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 it is always illegal combination.
* 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

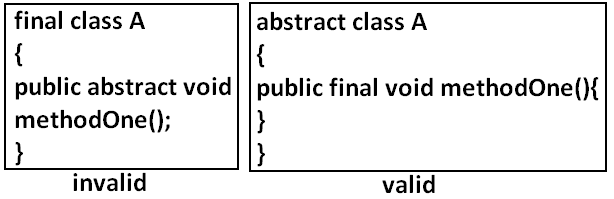
class child extends Parent

* 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:**



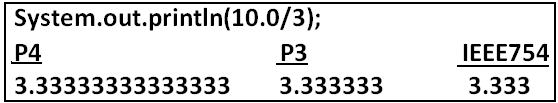
**Note:**

* 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.
* 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 flat from independent results.

**Example:**

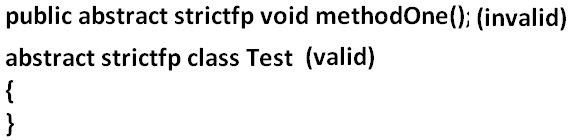


* If a class declares as the Strictfp then every concrete method(which has body) of that class has to follow IEEE754 standard for floating point arithmetic.

**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.

**Example:**



**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 with in 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.

**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 language.

**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

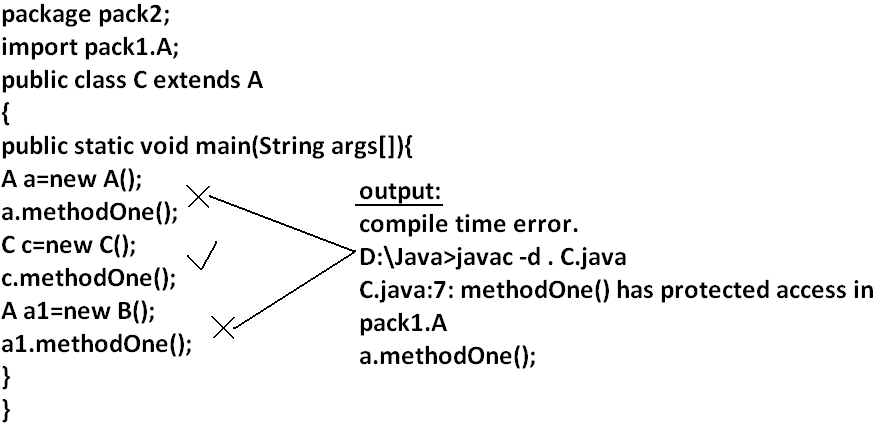
D:\Java>java pack1.B

methodOne is executed

methodOne is executed

methodOne is executed

**Example 2:**



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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **visibility** | **private** | **default** | **protected** | **public** |
| **1)With in the same class** |  |  |  |  |
| **2)From child class of same package** |  |  |  |  |
| **3)From non-child class of same package** |  |  |  |  |
| **4)From child class of outside package** |  |  |  |  |
| **5)From non-child class of outside package** |  |  |  |  |

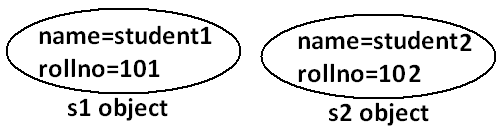
* The least accessible modifier is private.
* The most accessible modifier is public.
* Private<default<protected<public.
* Recommended modifier for variables is private where as recommended modifier for methods is public.

**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 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>

1. **Inside instance block:**

**Example:**

class Test

{

final int i;

{

i=10;

}}

**Output:**

D:\Java>javac Test.java

D:\Java>

1. **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.**
* 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:**

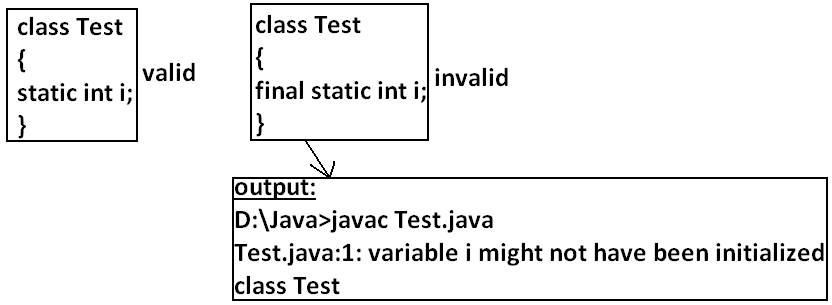
D:\Java>javac Test.java

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.

**Example:**



**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>

1. **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:**

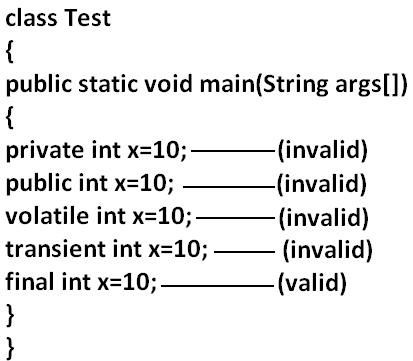
D:\Java>javac Test.java

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

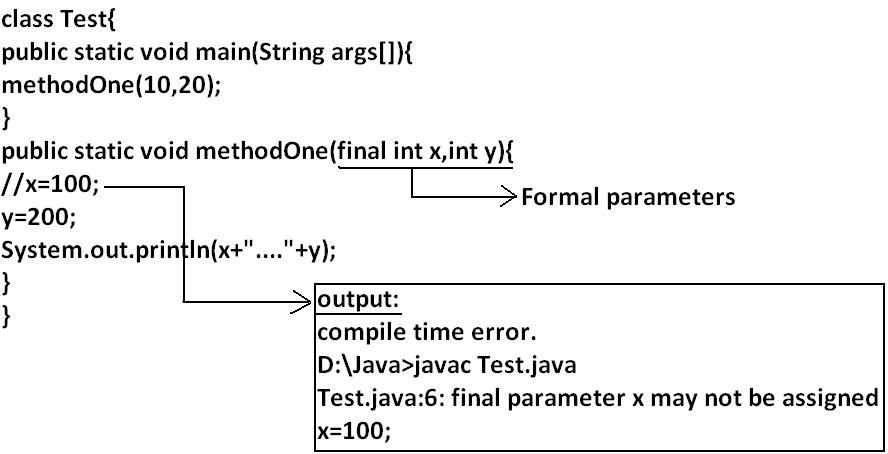
Test.java:5: illegal start of expression

private int x=10;

**Formal parameters:**

* The formal parameters of a method are simply access 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.

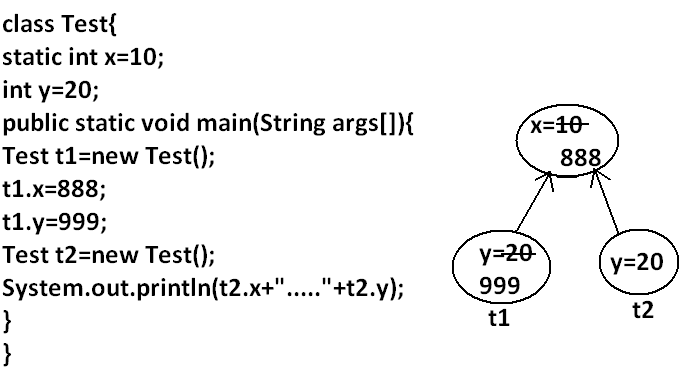
**Example:**



**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.

**Example:**



**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.

1. **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);

1. **2 and 3**

**Example:**

class Test

{

static int x=10;

public void methodOne(){

System.out.println(x);

}}

**Output:**

Compile successfully.

1. **2 and 4**

**Example:**

class Test

{

static int x=10;

public static void methodOne(){

System.out.println(x);

}}

**Output:**

Compile successfully.

1. **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;

1. **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.

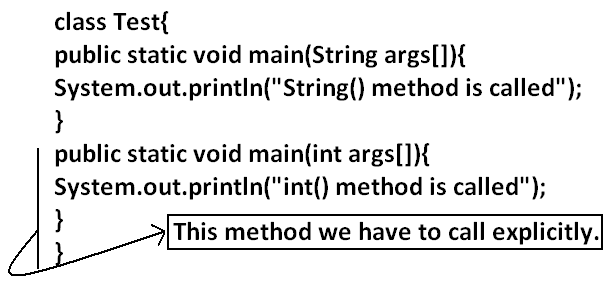
D:\Java>javac Test.java

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

public static void methodOne(){

* Overloading concept is applicable for static method including main method also.

**Example:**



* 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");

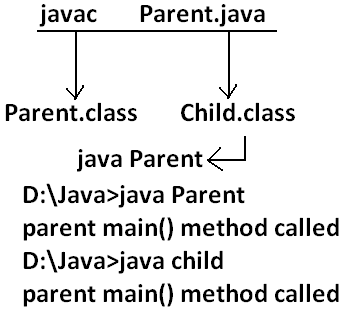
}

}

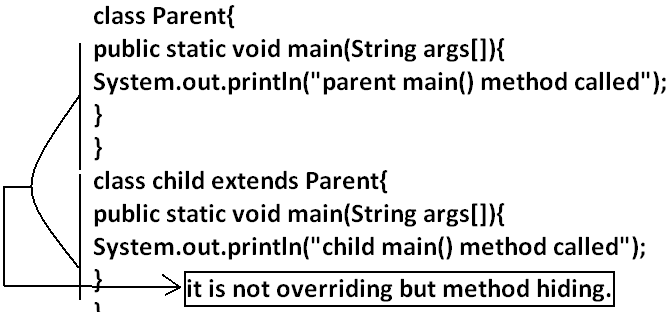
class child extends Parent{

}

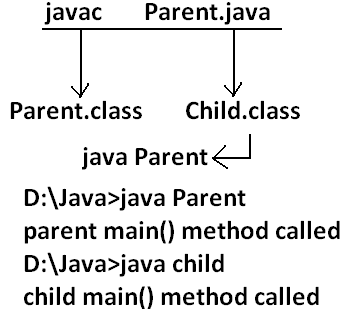
**Output:**



**Example:**



**Output:**

****

* It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.
* For static methods compulsory implementation should be available where as for abstract methods implementation should be available **hence abstract static combination is illegal for methods.**

**Native modifier:**

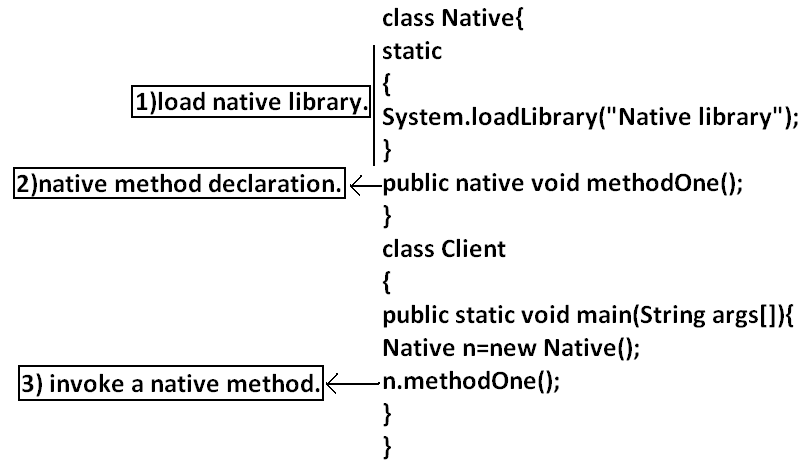
* Native is a modifier applicable only for methods but not for variables and classes.
* The methods which are implemented in non java are called native methods or foreign methods.

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

* To improve performance of the system.
* To use already existing legacy non java code.

**To 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 disadvantage of native keyword is usage of native keyword in java breaks platform independent nature of java language.

**Synchronized:**

* Synchronized is the modifier applicable for methods and blocks but not for variables and classes.
* 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.
* The main advantage of synchronized keyword is we can resolve data inconsistency problems, 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.

**Transient modifier:**

* Transient is the modifier applicable only for variables but not for methods and classes.
* 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.
* 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”.
* 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.
* Final variables will be participated into serialization directly by their values due to this declaring a final variable as transient there is no impact.

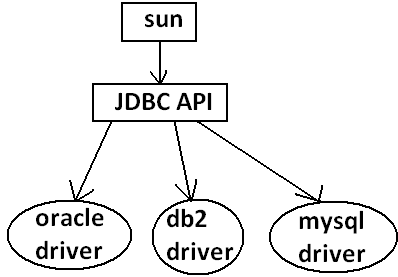
**Interfaces:**

1. Introduction
2. Interface declarations and implementations.
3. Extends vs implements
4. Interface methods
5. Interface variables
6. Interface naming conflicts
7. Method naming conflicts
8. Variable naming conflicts
9. Marker interface
10. Adapter class
11. Interface vs abstract class vs concrete class.
12. Difference between interface and abstract class?
13. Conclusions

**Def1:** Any service requirement specification (srs) is called an interface.

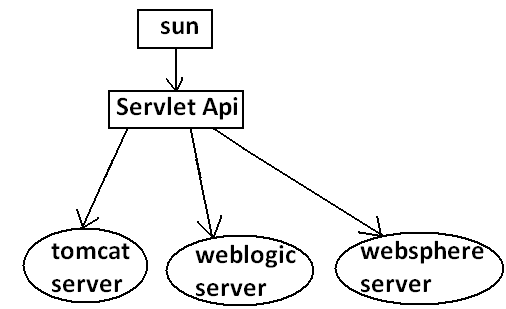
**Example1:** Sun people responsible to define JDBC API and database vendor will provide implementation for that.

**Diagram:**



**Example2:** Sun people define SERVLET API to develop web applications web server vendor is responsible to provide implementation.

**Diagram:**



**Def2:** From the client point of view an interface define the set of services what his excepting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.

**Example:** ATM GUI screen describes the set of services what bank people offering, at the same time the same GUI screen the set of services what customer his excepting hence this GUI screen acts as a contract between bank and customer.

**Def3:** Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

**Summery def:** Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

**Declaration and implementation of an interface:**

*Note1*: Whenever we are implementing an interface **compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract** in that case **child class is responsible to provide implementation for remaining methods**.

*Note2*: Whenever we are implementing an interface method **compulsory it should be declared as public otherwise we will get compile time error**.

**Example:**

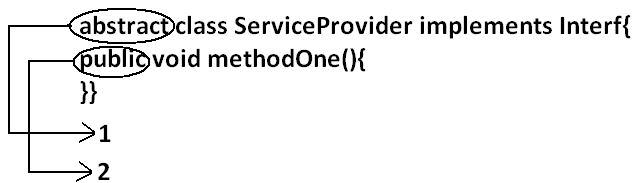
interface Interf

{

void methodOne();

void methodTwo();

}



class SubServiceProvider extends ServiceProvider

{

}

**Output:**

Compile time error.

D:\Java>javac SubServiceProvider.java

SubServiceProvider.java:1: SubServiceProvider is not abstract and does not override abstract method methodTwo() in Interf

class SubServiceProvider extends ServiceProvider

**Extends vs implements:**

* A class can extend only one class at a time.

**Example:**

class One{

public void methodOne(){

}

}

class Two extends One{

}

* A class can implements any no. Of interfaces at a time.

**Example:**

interface One{

public void methodOne();

}

interface Two{

public void methodTwo();

}

class Three implements One,Two{

public void methodOne(){

}

public void methodTwo(){

}

}

* A class can extend a class and can implement an interface simultaneously.

interface One{

void methodOne();

}

class Two

{

public void methodTwo(){

}

}

class Three extends Two implements One{

public void methodOne(){

}

}

* An interface can extend any no. Of interfaces at a time.

**Example:**

interface One{

void methodOne();

}

interface Two{

void methodTwo();

}

interface Three extends One,Two

{

}

1. **Which of the following is true?**
2. A class can extend any no.Of classes at a time.
3. An interface can extend only one interface at a time.
4. A class can implement only one interface at a time.
5. A class can extend a class and can implement an interface but not both simultaneously.
6. None of the above.

Ans: 5

1. Consider the expression **X extends Y** for which of the possibility of X and Y this expression is true?
2. Both x and y should be classes.
3. Both x and y should be interfaces.
4. Both x and y can be classes or can be interfaces.
5. No restriction.

Ans: 3

1. **X extends Y, Z?**

* X, Y, Z should be interfaces.

1. **X extends Y implements Z?**

* X, Y should be classes.
* Z should be interface.

1. **X implements Y, Z?**

* X should be class.
* Y, Z should be interfaces.

1. **X implements Y extend Z?**

**Example:**

interface One{

}

class Two {

}

class Three implements One extends Two{

}

**Output:**

Compile time error.

D:\Java>javac Three.java

Three.java:5: '{' expected

class Three implements One extends Two{

* Every method present inside interface is always **public and abstract** whether we are declaring or not. Hence inside interface the following method declarations are equal.

void methodOne();

public Void methodOne();

abstract Void methodOne(); **Equal**

public abstract Void methodOne();

* As every interface method is always public and abstract we can’t use the following modifiers for interface methods.
* **Private, protected, final, static, synchronized, native, strictfp.**

**Inside interface which method declarations are valid?**

1. public void methodOne(){}
2. private void methodOne();
3. public final void methodOne();
4. public static void methodOne();
5. public abstract void methodOne();

Ans: 5

**Interface variables:**

* An interface can contain variables to define requirement level constants.
* Every interface variable is always **public static and final** whether we are declaring or not.

**Example:**

interface interf

{

int x=10;

}

**Public:** To make it available for every implementation class.

**Static:** Without existing object also we have to access this variable.

**Final:** Implementation class can access this value but cannot modify.

* Hence inside interface the following declarations are equal.

int x=10;

public int x=10;

static int x=10;

final int x=10; **Equal**

public static int x=10;

public final int x=10;

static final int x=10;

public static final int x=10;

* As every interface variable by default **public static final** we can’t declare with the following modifiers.
* Private
* Protected
* Transient
* Volatile
* For the interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.

**Example:**

interface Interf

{

int x;

}

**Output:**

Compile time error.

D:\Java>javac Interf.java

Interf.java:3: = expected

int x;

**Which of the following declarations are valid inside interface?**

1. int x;
2. private int x=10;
3. public volatile int x=10;
4. public transient int x=10;
5. public static final int x=10;

Ans: 5

* Interface variables can be access from implementation class but cannot be modified.

**Example:**

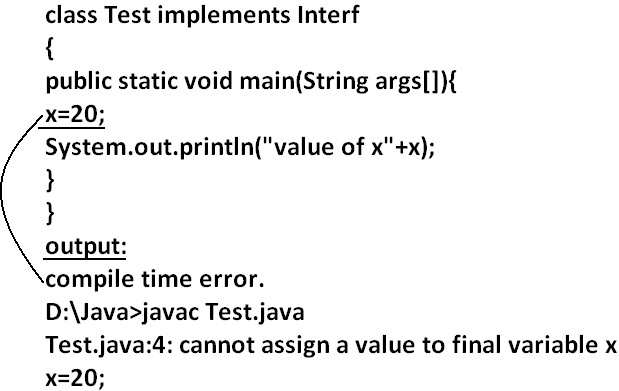
interface Interf

{

int x=10;

}

**Example 1:**



**Example 2:**

class Test implements Interf

{

public static void main(String args[]){

int x=20;

//here we declaring the variable x.

System.out.println(x);

}

}

**Output:**

D:\Java>javac Test.java

D:\Java>java Test

20

**Interface naming conflicts:**

**Method naming conflicts:**

**Case 1:**

* If two interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

**Example 1:**

interface Left

{

public void methodOne();

}

**Example 2:**

interface Right

{

public void methodOne();

}

**Example 3:**

class Test implements Left,Right

{

public void methodOne()

{

}}

**Output:**

D:\Java>javac Left.java

D:\Java>javac Right.java

D:\Java>javac Test.java

**Case 2:**

* if two interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods acts as a overloaded methods

**Example 1:**

interface Left

{

public void methodOne();

}

**Example 2:**

interface Right

{

public void methodOne(int i);

}

**Example 3:**

class Test implements Left,Right

{

public void methodOne()

{

}

public void methodOne(int i)

{

}}

**Output:**

D:\Java>javac Left.java

D:\Java>javac Right.java

D:\Java>javac Test.java

**Case 3:**

* If two interfaces contain a method with same signature but different return types then it is not possible to implement both interfaces simultaneously.

**Example 1:**

interface Left

{

public void methodOne();

}

**Example 2:**

interface Right

{

public int methodOne(int i);

}

* We can’t write any java class that implements both interfaces simultaneously.

Is a java class can implement any no. Of interfaces simultaneously?

* Yes, except if two interfaces contains a method with same signature but different return types.

**Variable naming conflicts:**

* Two interfaces can contain a variable with the same name and there may be a chance variable naming conflicts but we can resolve variable naming conflicts by using interface names.

**Example 1:**

interface Left

{

int x=888;

}

**Example 2:**

interface Right

{

int x=999;

}

**Example 3:**

class Test implements Left,Right

{

public static void main(String args[]){

//System.out.println(x);

System.out.println(Left.x);

System.out.println(Right.x);

}

}

**Output:**

D:\Java>javac Left.java

D:\Java>javac Right.java

D:\Java>javac Test.java

D:\Java>java Test

888

999

**Marker interface:** if an interface doesn’t contain any methods and by implementing that interface if our object gets some ability such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.

**Example:**

Serilizable

cloneable

RandomAccess These are marked for some ability

SingleThreadModel

.

.

.

.

**Example 1:** By implementing Serilizable interface we can send that object across the network and we can save state of an object into a file.

**Example 2:** By implementing SingleThreadModel interface Servlet can process only one client request at a time so that we can get “Thread Safety”.

**Example 3:** By implementing Cloneable interface our object is in a position to provide exactly duplicate cloned object.

**Without having any methods in marker interface how objects will get ability?**

* Internally JVM will provide required ability.

**Why JVM is providing the required ability?**

* To reduce complexity of the programming.

**Is it possible to create our own marker interface?**

* Yes, but customization of JVM is required.

**Adapter class:**

* Adapter class is a simple java class that implements an interface only with empty implementation for every method.
* If we implement an interface directly for each and every method compulsory we should provide implementation whether it is required or not. This approach increases length of the code and reduces readability.

**Example 1:**

interface X{

void m1();

void m2();

void m3();

void m4();

//.

//.

//.

//.

void m5();

}

**Example 2:**

class Test implements X{

public void m3(){

System.out.println("m3() method is called");

}

public void m1(){}

public void m2(){}

public void m4(){}

public void m5(){}

}

* We can resolve this problem by using adapter class.
* Instead of implementing an interface if we can extend adapter class we have to provide implementation only for required methods but not for all methods of that interface.
* This approach **decreases length of the code** and improves readability.

**Example 1:**

abstract class AdapterX implements X{

public void m1(){}

public void m2(){}

public void m3(){}

public void m4(){}

//.

//.

//.

public void m1000(){}

}

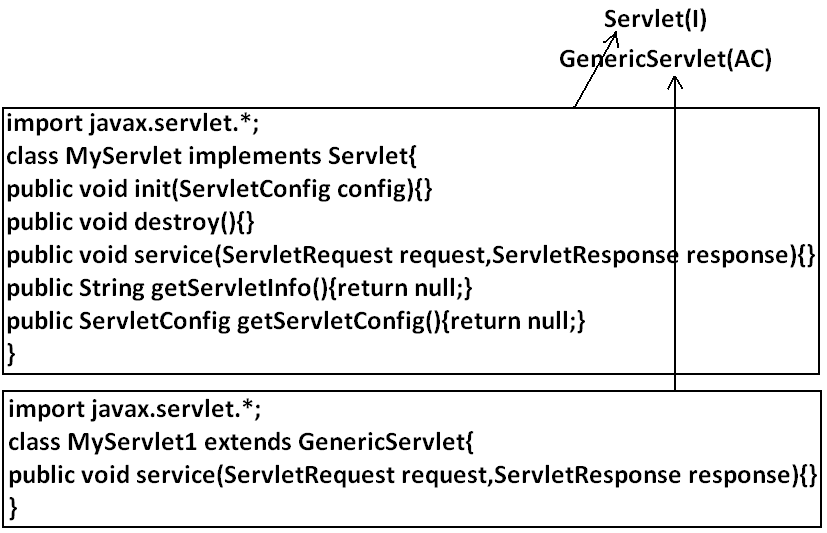
**Example 2:**

public class Test extend AdapterX{{

public void m3(){

}}

**Example:**



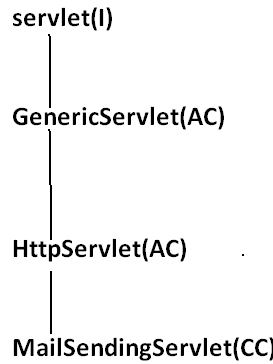
* Generic Servlet simply acts as an adapter class for Servlet interface.

**What is the difference between interface, abstract class and concrete class?**

**When we should go for interface, abstract class and concrete class?**

* If we don’t know anything about implementation just we have requirement specification then we should go for interface.
* If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
* If we are talking about implementation completely and ready to provide service then we should go for concrete class.

**Example:**

****

**What is the Difference between interface and abstract class?**

|  |  |
| --- | --- |
| interface | Abstract class |
| 1. If we don’t’ know anything about implementation just we have requirement specification then we should go for interface. | 1. If we are talking about implementation but not completely (partial implementation) then we should go for abstract class. |
| 1. Every method present inside interface is always **public** **and abstract** whether we are declaring or not. | 1. Every method present inside abstract class **need not be public and abstract**. |
| 1. We can’t declare interface methods with the modifiers **private**, **protected**, **final**, **static**, **synchronized**, **native**, **strictfp**. | 1. There are no restrictions on abstract class method modifiers. |
| 1. Every interface variable is always **public static final** whether we are declaring or not. | 1. Every abstract class variable need not be public static final. |
| 1. Every interface variable is always **public static** **final** we can’t declare with the following modifiers. **Private, protected, transient, volatile.** | 1. There are no restrictions on abstract class variable modifiers. |
| 1. For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error. | 1. It is not require to perform initialization for abstract class variables at the time of declaration. |
| 1. Inside interface we can’t take static and instance blocks. | 1. Inside abstract class we can take both static and instance blocks. |
| 1. Inside interface we can’t take constructor. | 1. Inside abstract class we can take constructor. |

**We can’t create object for abstract class but abstract class can contain constructor what is the need?**

* This constructor will be executed for the initialization of child object.

**Example:**

class Parent{

Parent()

{

System.out.println(this.hashCode());

}

}

class child extends Parent{

child(){

System.out.println(this.hashCode());

}

}

class Test{

public static void main(String args[]){

child c=new child();

System.out.println(c.hashCode());

}

}

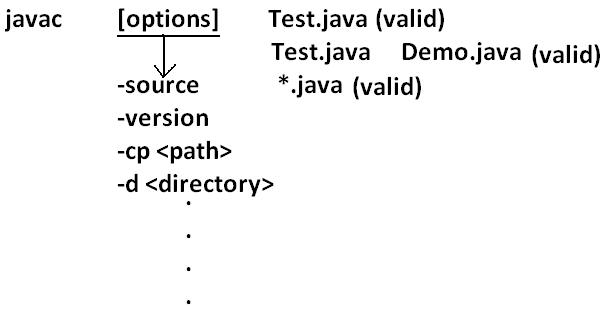
**Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept?**

* We can replace interface concept with abstract class. But it is not a good programming practice. We are misusing the roll of abstract class.

**Development**

**Javac:** we can use Javac to compile a single or group of “.java files”.

**Syntax**:



**Java:** we can use java command to run a single “.class file”.

**Syntax:**



**Classpath:** Class path describes the location where the required “.class files” are available. We can set the class path in the following 3 ways.

1. Permanently by using environment variable “classpath”. This class path will be preserved after system restart also.
2. Temporary for a particular command prompt level by using “set” command.

**Example:**



* Once if you close the command prompt automatically this class path will be lost.

1. We can set the class path for a particular command level by using “–cp” (or) “–class path”. This class path is applicable only for that command execution. After executing the command this classpath will be lost.

* Among the 3 ways of setting the class path the most common way is setting class path at command level by using “–cp”.

**Example 1:**

class Rain

{

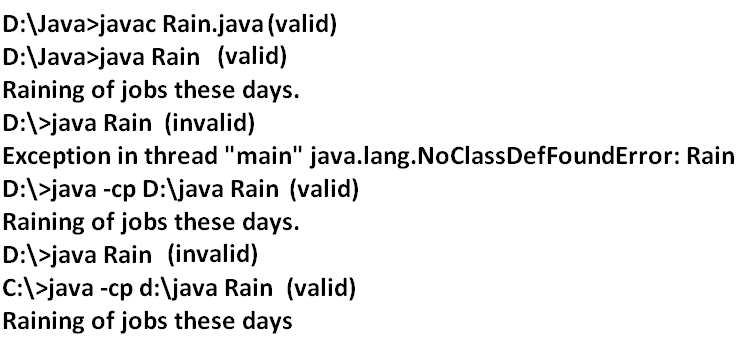
public static void main(String args[]){

System.out.println("Raining of jobs these days");

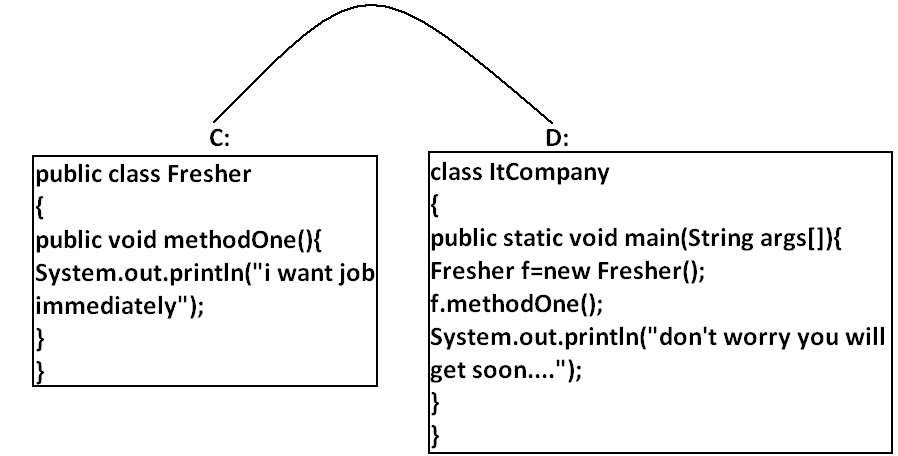
}

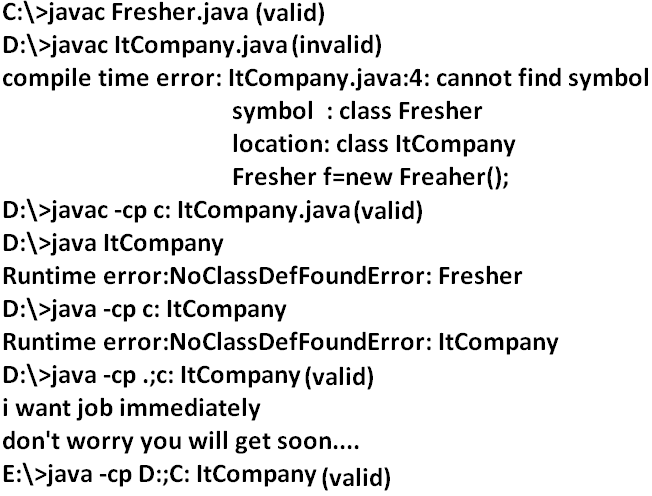
}

**Analysis:**

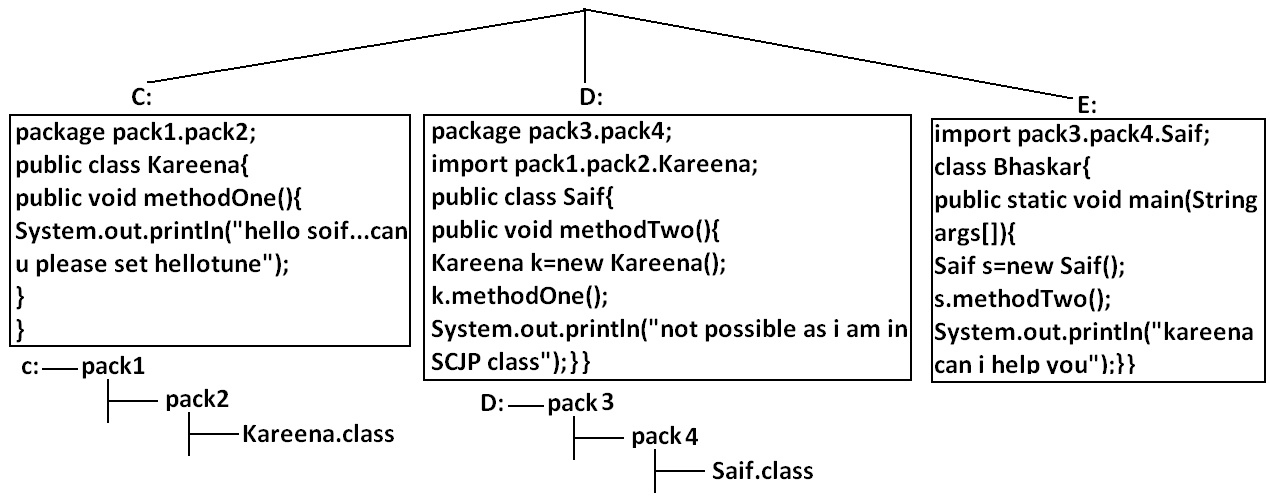


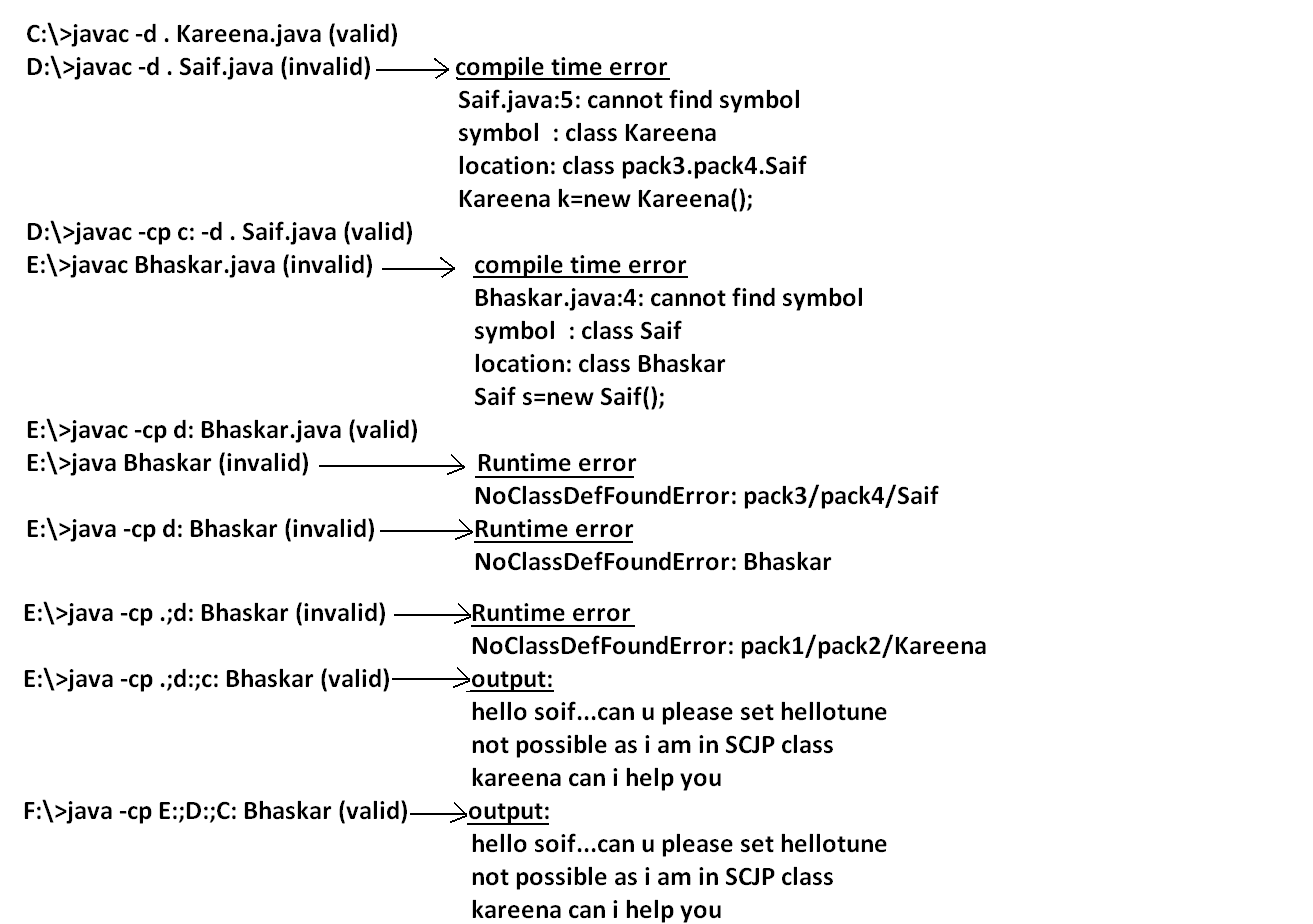
**Example 2:**

**Analysis:**



**Example 3:**

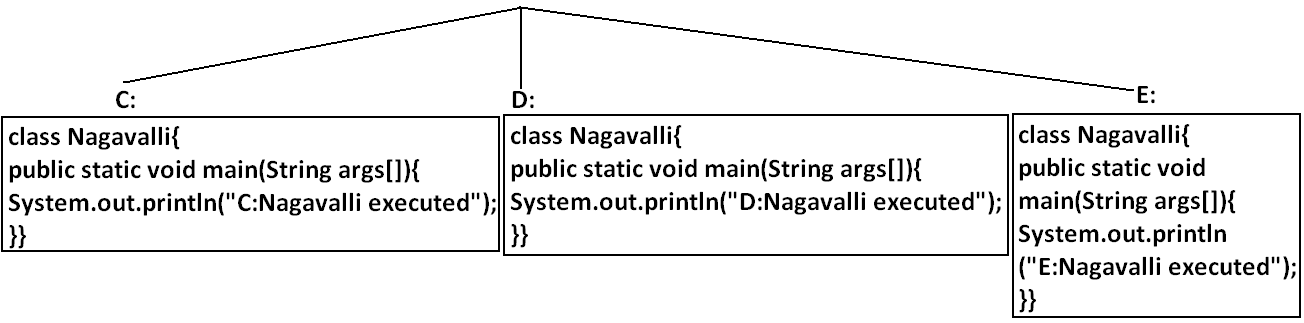
**Analysis:**

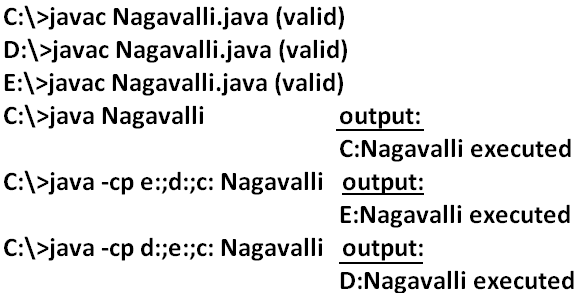


**Note:** if any folder structure created because of package statement. It should be resolved by import statement only and the location of base package should be make it available in class path.

**Note:** in classpath the order of locations is very important and it should be from left to right.

**Example 4:**

**Analysis:**

****

**Jar file:** If several dependent classes present then it is never recommended to set the classpath individual for every component. We have to group all these “.class files” into a single jar file and we have to make that jar file available to the classpath.

**Example:** All required classes to develop a Servlet are grouped into a single jar file (Servlet-api.jar) hence while compiling Servlet classes we have to make this jar file available in the classpath.

What is the difference between Jar, War and Ear?

**Jar (java archive):** Represents a group of “.class files”.

**War (web archive):** Represents a web application which may contains Servlets, JSP, HTML pages, JavaScript files etc.

**Ear (Enterprise archive):** it represents an enterprise application which may contain Servlets, JSP, EJB’S, JMS component etc.

* In generally an ear file consists of a group of war files and jar files.

Ear=war+ jar

**Various Commands:**

**To create a jar file:**

D:\Enum>jar -cvf bhaskar.jar Beer.class Test.class X.class

D:\Enum>jar -cvf bhaskar.jar \*.class

**To extract a jar file:**

D:\Enum>jar -xvf bhaskar.jar

**To display table of contents of a jar file:**

D:\Enum>jar -tvf bhaskar.jar

**Example 5:**

public class BhaskarColorFulCalc{

public static int add(int x,int y){

return x\*y;

}

public static int multiply(int x,int y){

return 2\*x\*y;

}}

**Analysis:**

C:\>javac BhaskarColorFulCalc.java

C:\>jar -cvf bhaskar.jar BhaskarColorFulCalc.class

**Example 6:**

class Client{

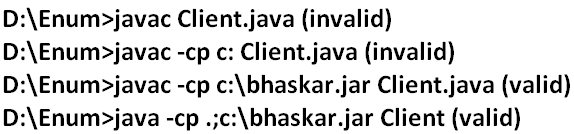
public static void main(String args[]){

System.out.println(BhaskarColorFulCalc.add(10,20));

System.out.println(BhaskarColorFulCalc.multiply(10,20));

}}

**Analysis:**



**Note:** Whenever we are placing jar file in the classpath compulsory we have to specify the name of the jar file also and just location is not enough.

**System properties:**

* For every system some persistence information is available in the form of system properties. These may include name of the os, java version, vendor of jvm etc.
* We can get system properties by using getProperties() method of system class. The following program displays all the system properties.

**Example 7:**

import java.util.\*;

class Test{

public static void main(String args[]){

//Properties is a class in util package.

//here getPropertes() method returns the Properties object.

Properties p=System.getProperties();

p.list(System.out);

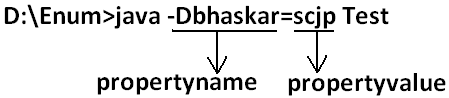
}

}

**How to set system property from the command prompt:**

* We can set system property from the command prompt by using –D option.

**Command:**



**What is the difference between path and classpath?**

**Path:** We can use “path variable” to specify the location where required binary executables are available.

* If we are not setting path then “java” and “Javac” commands won’t work.

**Classpath:** We can use “classpath variable” to describe location where required class files are available.

* If we are not setting classpath then our program won’t compile and run.

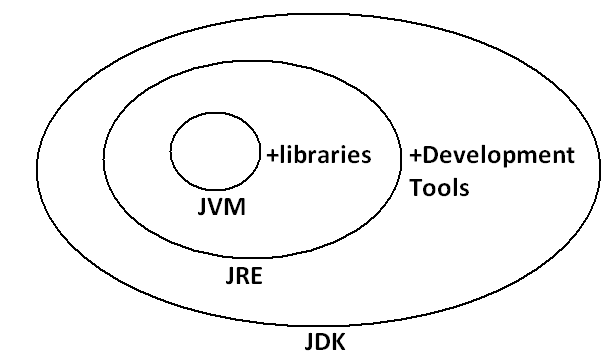
**What is the difference between JDK, JRE and JVM?**

**JDK (java development kit):** To develop and run java applications the required environment is JDK.

**JRE (java runtime environment):** To run java application the required environment is JRE.

**JVM (java virtual machine):** To execute java application the required virtual machine is JVM.

**Diagram:**



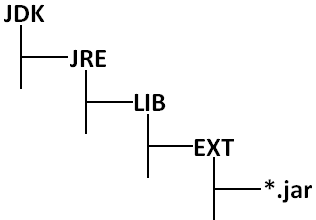
* JDK=JRE+Development Tools.
* JRE=JVM+Libraries.
* JRE is the part of JDK.
* Jvm is the part of JRE.

Note: At client side JRE is required and at developers side JDK is required.

**Shortcut way to place a jar files:**

* If we are placing jar file in the following location then it is not required to set classpath explicitly.

**Diagram:**



**Garbage Collection**:

1. Introduction:
2. The way to make an object eligible for GC
3. The methods for requesting JVM to run GC
4. Finalization

**Introduction:**

* In old languages like C++ programmer is responsible for both creation and destruction of objects. Usually programmer is taking very much care while creating object and neglect destruction of useless objects .Due to his negligence at certain point of time for creation of new object sufficient memory may not be available and entire application may be crashed due to memory problems.
* But in java programmer is responsible only for creation of new object and his not responsible for destruction of objects.
* Sun people provided one assistant which is always running in the background for destruction at useless objects. Due to this assistant the chance of failing java program is very rare because of memory problems.
* This assistant is nothing but garbage collector. Hence the main objective of GC is to destroy useless objects.

**The ways to make an object eligible for GC:**

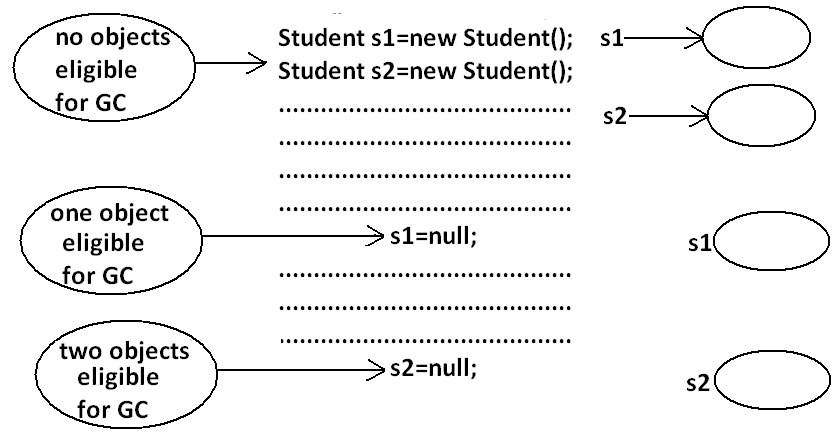
* Even through programmer is not responsible for destruction of objects but it is always a good programming practice to make an object eligible for GC if it is no longer required.
* An object is eligible for GC if and only if it does not have any references.

**The following are various possible ways to make an object eligible for GC:**

**1. Nullifying the reference variable:**

* If an object is no longer required then we can make eligible for GC by assigning “null” to all its reference variables.

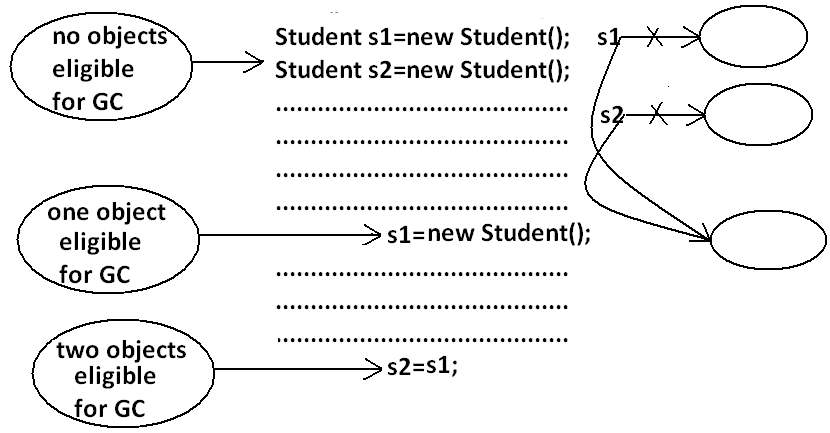
**Example:**



**2. Reassign the reference variable:**

* If an object is no longer required then reassign all its reference variables to some other objects then old object is by default eligible for GC.

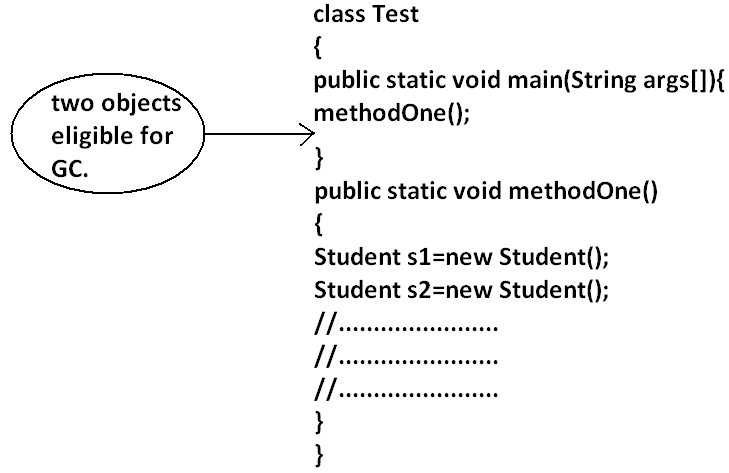
**Example:**



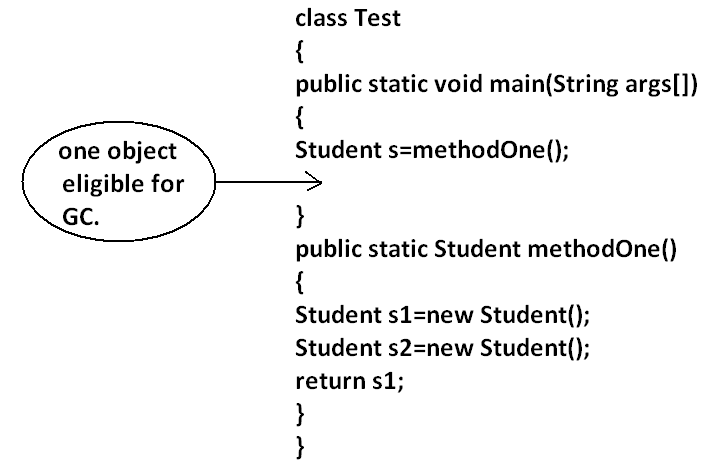
**3. Objects created inside a method:**

* Objects created inside a method are by default eligible for GC once method completes.

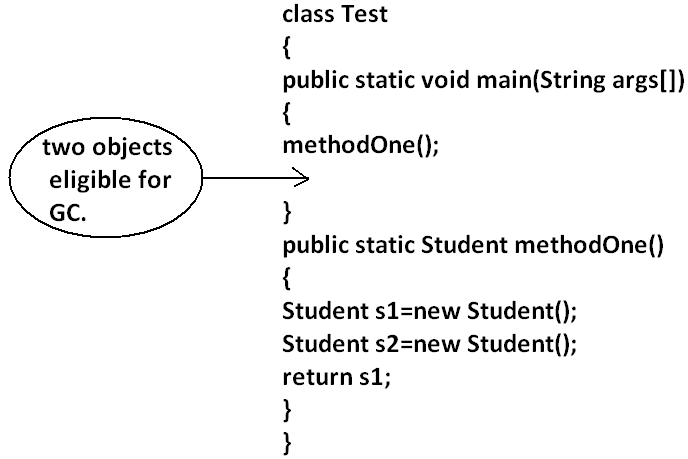
**Example 1:**



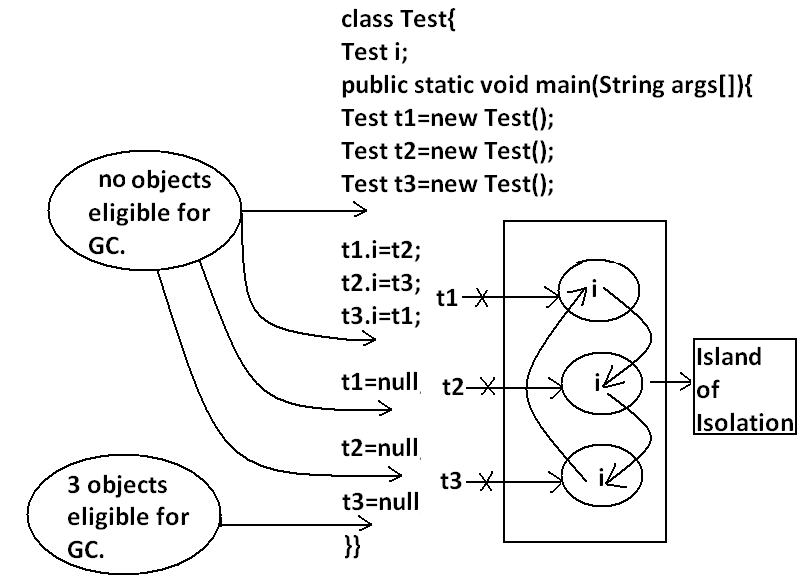
**Example 2:**



**Example 3:**



**Island of Isolation:**



**Note:** if an object doesn’t have any reference then it always eligible for GC.

**Note:** Even though object having reference still it is eligible for GC some times.

**Example:** island of isolation.

**The methods for requesting JVM to run GC:**

* Once we made an object eligible for GC it may not be destroyed immediately by the GC. Whenever jvm runs GC then only object will be destroyed by the GC. But when exactly JVM runs GC we can’t expert it is vendor dependent.
* We can request jvm to run garbage collector programmatically, but whether jvm accept our request or not there is no guaranty.

**The following are various ways for requesting jvm to run GC:**

**By System class:** System class contains a static method GC for this.

**Example:** System.gc();

**By Runtime class:** A java application can communicate with jvm by using Runtime object.

* Runtime class is a singleton class present in java.lang. Package. We can create Runtime object by using factory method getRuntime().

**Example:** Runtime r=Runtime.getRuntime();

* Once we got Runtime object we can call the following methods on that object.

**freeMemory():** returns the free memory present in the heap.

**totalMemory():** returns total memory of the heap.

**gc():** for requesting jvm to run gc.

**Example:**

import java.util.Date;

class RuntimeDemo

{

public static void main(String args[]){

Runtime r=Runtime.getRuntime();

System.out.println("total memory of the heap :"+r.totalMemory());

System.out.println("free memory of the heap :"+r.freeMemory());

for(int i=0;i<10000;i++)

{

Date d=new Date();

d=null;

}

System.out.println("free memory of the heap :"+r.freeMemory());

r.gc();

System.out.println("free memory of the heap :"+r.freeMemory());

}

}

**Output:**

Total memory of the heap: 5177344

Free memory of the heap: 4994920

Free memory of the heap: 4743408

Free memory of the heap: 5049776

**Which of the following are valid ways for requesting jvm to run GC?**

1. System.gc(); **(valid)**
2. Runtime.gc(); **(invalid)**
3. (new Runtime).gc(); **(invalid)**
4. Runtime.getRuntime().gc(); **(valid)**

**Note:** gc() method present in System class is static where as it is instance method in Runtime class.

**Note:** Over Runtime class gc() method System class gc() method is recommended to use.

**Note:** in java it is not possible to find size of an object and address of an object.

**Finalization:**

* Just before destroying any object gc always calls finalize() method to perform cleanup activities. If the corresponding class contains finalize() method then it will be executed otherwise Object class finalize() method will be executed which is declared as follows.

protected void **finalize**() throws [Throwabl](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\Throwable.html)e

**Case 1:**

* Just before destroying any object GC calls finalize() method on the object which is eligible for GC then the corresponding class finalize() method will be executed. For example if String object is eligible for GC then String class finalize()method is executed but not Test class finalize()method.

**Example:**

class Test

{

public static void main(String args[]){

String s=new String("bhaskar");

Test t=new Test();

s=null;

System.gc();

System.out.println("End of main.");

}

public void finalize(){

System.out.println("finalize() method is executed");

}

}

**Output:**

End of main.

* In the above program String class finalize()method got executed. Which has empty implementation.
* If we replace String object with Test object then Test class finalize() method will be executed .The following program is an example of this.

**Example:**

class Test

{

public static void main(String args[]){

String s=new String("bhaskar");

Test t=new Test();

t=null;

System.gc();

System.out.println("End of main.");

}

public void finalize(){

System.out.println("finalize() method is executed");

}

}

**Output:**

finalize() method is executed

End of main

**Case 2:**

* We can call finalize() method explicitly then it will be executed just like a normal method call and object won’t be destroyed. But before destroying any object GC always calls finalize() method.

**Example:**

class Test

{

public static void main(String args[]){

Test t=new Test();

t.finalize();

t.finalize();

t=null;

System.gc();

System.out.println("End of main.");

}

public void finalize(){

System.out.println("finalize() method called");

}

}

**Output:**

finalize() method called.

finalize() method called.

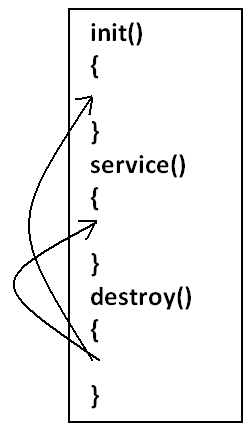
finalize() method called.

End of main.

* In the above program finalize() method got executed 3 times in that 2 times explicitly by the programmer and one time by the gc.

**Note:** In Servlets we can call destroy() method explicitly from init() and service() methods. Then it will be executed just like a normal method call and Servlet object won’t be destroyed.

**Diagram:**



**Case 3:**

* finalize() method can be call either by the programmer or by the GC . If the programmer calls explicitly finalize() method and while executing the finalize() method if an exception raised and uncaught then the program will be terminated abnormally.
* If GC calls finalize() method and while executing the finalize()method if an exception raised and uncaught then JVM simply ignores that exception and the program will be terminated normally.

**Example:**

class Test

{

public static void main(String args[]){

Test t=new Test();

//t.finalize();-------line(1)

t=null;

System.gc();

System.out.println("End of main.");

}

public void finalize(){

System.out.println("finalize() method called");

System.out.println(10/0);

}

* If we are not comment line1 then programmer calling finalize() method explicitly and while executing the finalize()method ArithmeticException raised which is uncaught hence the program terminated abnormally.
* If we are comment line1 then GC calls finalize() method and JVM ignores ArithmeticException and program will be terminated normally.

**Which of the following is true?**

1. While executing finalize() method JVM ignores every exception(invalid).
2. While executing finalize() method JVM ignores only uncaught exception(valid).

**Case 4:**

* On any object GC calls finalize() method only once.

**Example:**

class FinalizeDemo

{

static FinalizeDemo s;

public static void main(String args[])throws Exception{

FinalizeDemo f=new FinalizeDemo();

System.out.println(f.hashCode());

f=null;

System.gc();

Thread.sleep(5000);

System.out.println(s.hashCode());

s=null;

System.gc();

Thread.sleep(5000);

System.out.println("end of main method");

}

public void finalize()

{

System.out.println("finalize method called");

s=this;

}

}

**Output:**

D:\Enum>java FinalizeDemo

4072869

finalize method called

4072869

End of main method

**Note:**

* The behavior of the GC is vendor dependent and varied from JVM to JVM hence we can’t expert exact answer for the following.

1. What is the algorithm followed by GC.

2. Exactly at what time JVM runs GC.

3. In which order GC identifies the eligible objects.

4. In which order GC destroys the object etc.

**Memory leak:**

* An object which is not using in our application and it is not eligible for GC such type of objects are called “memory leaks”.
* In the case of memory leaks GC also can’t do anything the application will be crashed due to memory problems.
* By using monitoring tools we can identify memory leaks.

**Example:**

1. HPJ meter
2. HP ovo
3. IBM Tivoli These are monitoring tools.
4. J Probe
5. Patrol and etc

**ENUM**

* We can use enum to define a group of named constants.

**Example 1:**

enum Month

{

JAN,FEB,MAR,DEC;

}

**Example 2:**

enum Beer

{

KF,KO,RC,FO;

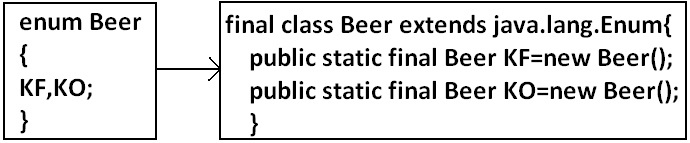
}

* Enum concept introduced in 1.5 versions.
* When compared with old languages enum java’s enum is more powerful.
* By using enum we can define our own data types which are also come enumerated data types.

**Internal implementation of enum:**

* Internally enum’s are implemented by using class concept. Every enum constant is a reference variable to that enum type object.
* Every enum constant is implicitly **public static final** always.

**Example 3:**



**Diagram:**



**Declaration and usage of enum:**

**Example 4:**

enum Beer

{

KF,KO,RC,FO;//here semicolon is optional.

}

class Test

{

public static void main(String args[]){

Beer b1=Beer.KF;

System.out.println(b1);

}

}

**Output:**

D:\Enum>java Test

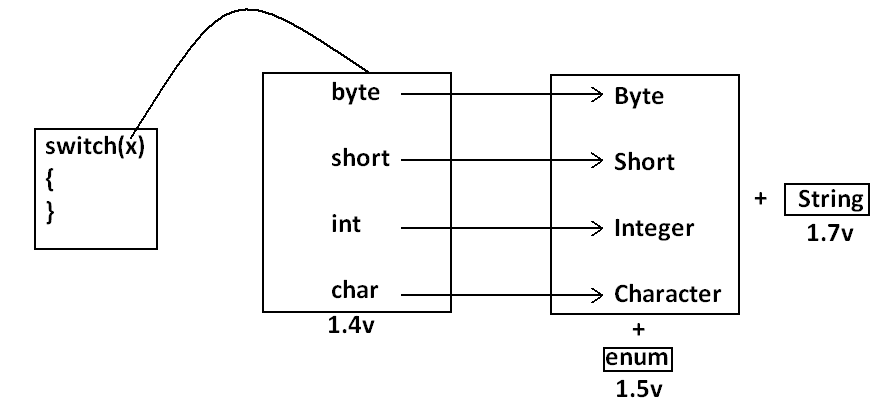
KF

Note: Every enum constant internally static hence we can access by using “enum name”.

**Enum vs switch statement:**

* Until 1.4 versions the allowed types for the switch statement are byte, short, char int. But from 1.5 version onwards in addition to this the corresponding wrapper classes and enum type also allowed. That is from 1.5 version onwards we can use enum type as argument to switch statement.

**Diagram:**



**Example:**

enum Beer

{

KF,KO,RC,FO;

}

class Test{

public static void main(String args[]){

Beer b1=Beer.RC;

switch(b1){

case KF:

System.out.println("it is childrens brand");

break;

case KO:

System.out.println("it is too lite");

break;

case RC:

System.out.println("it is too hot");

break;

case FO:

System.out.println("buy one get one");

break;

default:

System.out.println("other brands are not good");

}

}}

**Output:**

D:\Enum>java Test

It is too hot

* If we are passing enum type as argument to switch statement then every case label should be a valid enum constant otherwise we will get compile time error.

**Example:**

enum Beer

{

KF,KO,RC,FO;

}

class Test{

public static void main(String args[]){

Beer b1=Beer.RC;

switch(b1){

case KF:

case RC:

case KALYANI:

}}}

**Output:**

Compile time error.

D:\Enum>javac Test.java

Test.java:11: unqualified enumeration constant name required

case KALYANI:

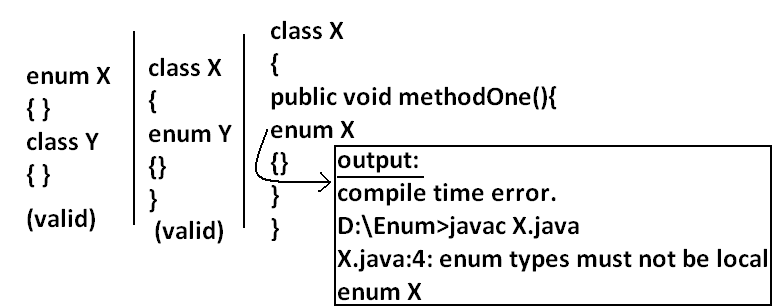
* We can declare enum either outside the class or within the class but not inside a method. If we declare enum outside the class the allowed modifiers are:

1. public
2. default
3. strictfp.

* If we declare enum inside a class then the allowed modifiers are:

1. public private
2. default + protected
3. strictfp static

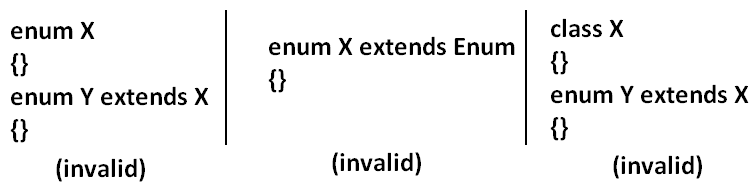
**Example:**



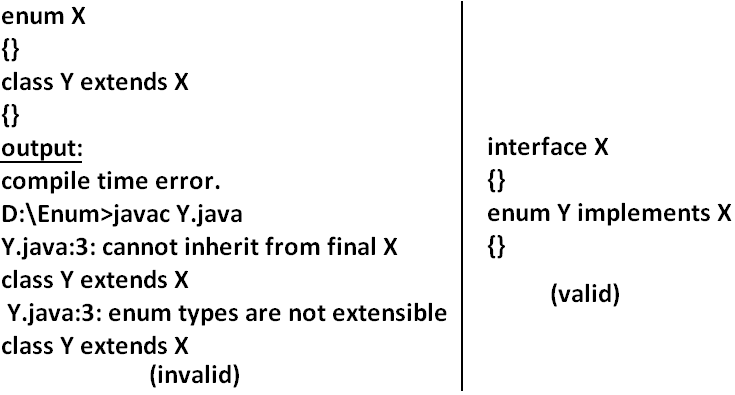
**Enum vs inheritance:**

* Every enum in java is the direct child class of java.lang.Enum class hence it is not possible to extends any other enum.
* Every enum is implicitly final hence we can’t create child enum.
* Because of above reasons we can conclude inheritance concept is not applicable for enum’s explicitly.
* But enum can implement any no. Of interfaces simultaneously.

**Example:**



**Example:**



**Java.lang.Enum:** Every enum in java is the direct child class of java.lang.Enum. The power of enum is inheriting from this class only.

## It is abstract class and it is direct child class of “Object class” it implements Serializable and Comparable.

**values() method:** Every enum implicitly contains a static values() method to list all constants of enum.

**Example:** Beer[] b=Beer.values();

**ordinal() method:** Within enum the order of constants is important we can specify by its ordinal value.

* We can find ordinal value(index value) of enum constant by using ordinal() method.

**Example:** public int ordinal();

**Example:**

enum Beer

{

KF,KO,RC,FO;

}

class Test{

public static void main(String args[]){

Beer[] b=Beer.values();

for(Beer b1:b)//this is forEach loop.

{

System.out.println(b1+"......."+b1.ordinal());

}}}

**Output:**

D:\Enum>java Test

KF.......0

KO.......1

RC.......2

FO.......3

**Specialty of java enum:** When compared with old languages enum java’s enum is more powerful because in addition to constants we can take normal variables, constructors, methods etc which may not possible in old languages.

* Inside enum we can declare main method and even we can invoke enum directly from the command prompt.

**Example:**

enum Fish{

GOLD,APOLO,STAR;

public static void main(String args[]){

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

}}

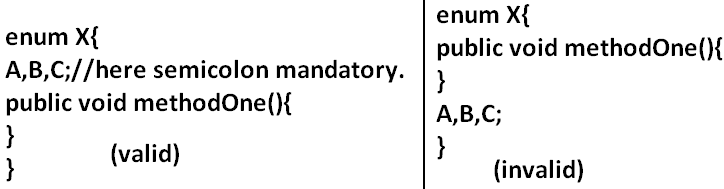
**Output:**

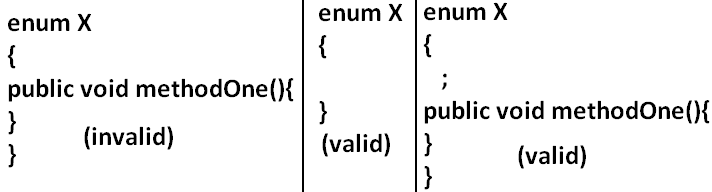
D:\Enum>java Fish

enum main() method called

* In addition to constants if we are taking any extra members like methods then the list of constants should be in the 1st line and should ends with semicolon.
* If we are taking any extra member then enum should contain at least one constant. Any way an empty enum is always valid.

**Example:**





**Enum vs constructor:** Enum can contain constructor. Every enum constant represents an object of that enum class which is static hence all enum constants will be created at the time of class loading automatically and hence constructor will be executed at the time of enum class loading for every enum constants.

**Example:**

enum Beer{

KF,KO,RC,FO;

Beer(){

System.out.println("Constructor called.");

}

}

class Test{

public static void main(String args[]){

Beer b=Beer.KF;

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

}}

**Output:**

D:\Enum>java Test

Constructor called.

Constructor called.

Constructor called.

Constructor called.

Hello.

* We can’t create enum object explicitly and hence we can’t invoke constructor directly.

**Example:**

enum Beer{

KF,KO,RC,FO;

Beer(){

System.out.println("constructor called");

}

}

class Test{

public static void main(String args[]){

Beer b=new Beer();

System.out.println(b);

}}

**Output:**

Compile time error.

D:\Enum>javac Test.java

Test.java:9: enum types may not be instantiated

Beer b=new Beer();

**Example:**



enum Beer

{

KF(100),KO(70),RC(65),Fo(90),KALYANI;

int price;

Beer(int price){

this.price=price;

}

Beer()

{

this.price=125;

}

public int getPrice()

{

return price;

}

}

class Test{

public static void main(String args[]){

Beer[] b=Beer.values();

for(Beer b1:b)

{

System.out.println(b1+"......."+b1.getPrice());

}}}

* Inside enum we can take both instance and static methods but it is not possible to take abstract methods.

**Case 1:**

* Every enum constant represents an object hence whatever the methods we can apply on the normal objects we can apply the same methods on enum constants also.

**Which of the following expressions are valid?**

1. Beer.KF==Beer.RC----------------------------> false
2. Beer.KF.equals(Beer.RC) ------------------->false
3. Beer.KF<Beer.RC------------------------------>invalid
4. Beer.KF.ordinal()<Beer.RC.ordinal()------>valid

**Case 2:**

**Example 1:**

package pack1;

public enum Fish

{

STAR,GUPPY;

}

**Example 2:**

package pack2;

//import static pack1.Fish.\*;

import static pack1.Fish.STAR;

class A

{

public static void main(String args[]){

System.out.println(STAR);

}

}

1. Import pack1.\*; ---------------------------->invalid
2. Import pack1.Fish; ------------------------->invalid
3. import static pack1.Fish.\*; --------------->valid
4. import static pack1.Fish.STAR; ---------->valid

**Example 3:**

package pack3;

//import pack1.Fish;

import pack1.\*;

//import static pack1.Fish.GUPPY;

import static pack1.Fish.\*;

class B

{

public static void main(String args[]){

Fish f=Fish.STAR;

System.out.println(GUPPY);

}

}

**Case 3:**

enum Color

{

BLUE,RED

{

public void info(){

System.out.println("Dangerous color");

}

},GREEN;

public void info()

{

System.out.println("Universal color");

}

}

class Test

{

public static void main(String args[]){

Color[] c=Color.values();

for(Color c1:c)

{

c1.info();

}}}

**Output:**

Universal color

Dangerous color

Universal color

**Internationalization**

* The process of designing a web application such that it supports various countries, various languages without performing any changes in the application is called Internationalization.
* We can implement Internationalization by using the following classes. They are:

1. Locale
2. NumberFormat
3. DateFormat

**1. Locale:** A Locale object can be used to represent a geographic (country) location (or) language.

* Locale class present in **java.util package**.
* It is a final class and direct child class of Object implements [Cloneable](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\Cloneable.html) and [Serializable](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\io\Serializable.html) Interfaces.

**How to create a Locale object:**

* We can create a Locale object by using the following constructors of Locale class.

1. Locale l=new Locale(String language);
2. Locale l=new Locale(String language,String country);

* Locale class already defines some predefined Locale constants. We can use these constants directly.

**Example:**

Locale. UK

Locale. US

Locale. ITALY

Locale. CHINA

**Important methods of Locale class:**

1. public static [Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html) getDefault()
2. public static void setDefault([Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html) l)
3. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) getLanguage()
4. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) getDisplayLanguage([Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html) l)
5. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) getCountry()
6. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) getDisplayCountry([Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html) l)
7. public static [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html)[] getISOLanguages()
8. public static [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html)[] getISOCountries()
9. public static [Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html)[] getAvailableLocales()

**Example for Locale:**

import java.util.\*;

class LocaleDemo{

public static void main(String args[]){

Locale l1=Locale.getDefault();

//System.out.println(l1.getCountry()+"....."+l1.getLanguage());

//System.out.println(l1.getDisplayCountry()+"....."+l1.getDisplayLanguage());

Locale l2=new Locale("pa","IN");

Locale.setDefault(l2);

String[] s3=Locale.getISOLanguages();

for(String s4:s3)

{

//System.out.print("ISO language is :");

//System.out.println(s4);

}

String[] s4=Locale.getISOCountries();

for(String s5:s4)

{

System.out.print("ISO Country is:");

System.out.println(s5);

}

Locale[] s=Locale.getAvailableLocales();

for(Locale s1:s)

{

//System.out.print("Available locales is:");

//System.out.println(s1.getDisplayCountry()+"......"+s1.getDisplayLanguage());

}}}

**NumberFormat:**

* Various countries follow various styles to represent number.

**Example:**

1,23,456.789------------INDIA

123,456.789-------------US

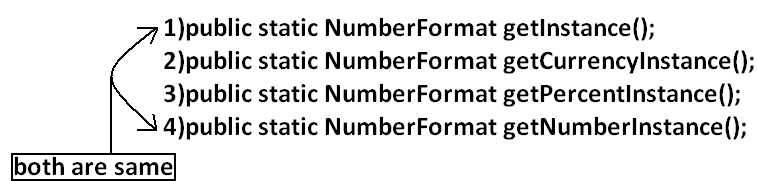
123.456,789-------------ITALY

* By using NumberFormat class we can format a number according to a particular Locale.
* NumberFormat class present in java.Text package and it is an abstract class.
* Hence we can’t create an object by using constructor.

NumberFormat nf=new NumberFormat(); --------invalid

**Getting NumberFormat object for the default Locale:**

* NumberFormat class defines the following methods for this.



**Getting NumberFormat object for the specific Locale:**

* The methods are exactly same but we have to pass the corresponding Locale object as argument.

**Example:** public static [NumberFormat](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\text\NumberFormat.html) getNumberInstance([Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html) l);

* Once we got NumberFormat object we can call the following methods to format and parse numbers.

1. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) format(long l);
2. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) format(double d);

* To convert a number from java form to Locale specific form.

1. public [Number](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\Number.html) parse([String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) source)throws [ParseException](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\text\ParseException.html)

* To convert from Locale specific String form to java specific form.

**Example:**

import java.util.\*;

import java.text.\*;

class NumberFormatDemo

{

public static void main(String args[]){

double d=123456.789;

NumberFormat nf=NumberFormat.getInstance(Locale.ITALY);

System.out.println("ITALY form is :"+nf.format(d));

}

}

**Output:**

ITALY form is :123.456,789

**Requirement:** Write a program to print a java number in INDIA, UK, US and ITALY currency formats.

**Program:**

import java.util.\*;

import java.text.\*;

class NumberFormatDemo

{

public static void main(String args[]){

double d=123456.789;

Locale INDIA=new Locale("pa","IN");

NumberFormat nf=NumberFormat.getCurrencyInstance(INDIA);

System.out.println("INDIA notation is :"+nf.format(d));

NumberFormat nf1=NumberFormat.getCurrencyInstance(Locale.UK);

System.out.println("UK notation is :"+nf1.format(d));

NumberFormat nf2=NumberFormat.getCurrencyInstance(Locale.US);

System.out.println("US notation is :"+nf2.format(d));

NumberFormat nf3=NumberFormat.getCurrencyInstance(Locale.ITALY);

System.out.println("ITALY notation is :"+nf3.format(d));

}}

**Output:**

INDIA notation is: INR 123,456.79

UK notation is: ú123,456.79

US notation is: $123,456.79

ITALY notation is: Ç 123.456,79

**Setting Maximum, Minimum, Fraction and Integer digits:**

* NumberFormat class defines the following methods for this purpose.

1. public void **setMaximumFractionDigits**(int n);
2. public void **setMinimumFractionDigits**(int n);
3. public void **setMaximumIntegerDigits**(int n);
4. public void **setMinimumIntegerDigits**(int n);

**Example:**

import java.text.\*;

public class NumberFormatExample

{

public static void main(String[] args){

NumberFormat nf=NumberFormat.getInstance();

nf.setMaximumFractionDigits(3);

System.out.println(nf.format(123.4));

System.out.println(nf.format(123.4567));

nf.setMinimumFractionDigits(3);

System.out.println(nf.format(123.4));

System.out.println(nf.format(123.4567));

nf.setMaximumIntegerDigits(3);

System.out.println(nf.format(1.234));

System.out.println(nf.format(123456.789));

nf.setMinimumIntegerDigits(3);

System.out.println(nf.format(1.234));

System.out.println(nf.format(123456.789));

}}

**Output:**

123.4

123.457

123.400

123.457

1.234

456.789

001.234

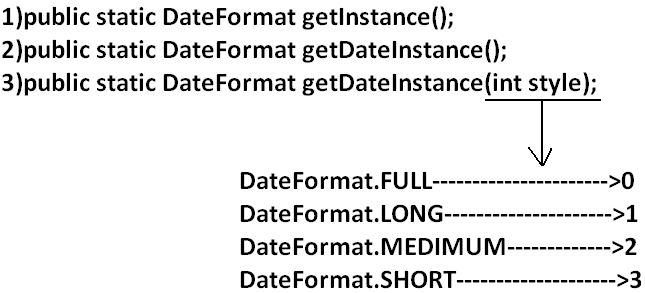
456.789

**DateFormat:** Various countries follow various styles to represent Date. We can format the date according to a particular locale by using DateFormat class.

* DateFormat class present in java.text package and it is an abstract class.

**Getting DateFormat object for default Locale:**

* DateFormat class defines the following methods for this purpose.



**Getting DateFormat object for the specific Locale:**

1. public static [DateFormat](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\text\DateFormat.html) getDateInstance(int style, [Locale](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Locale.html) l);

* Once we got DateFormat object we can format and parse Date by using the following methods.

1. public [String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) format([Date](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Date.html) date);

* To convert the date from java form to locale specific string form.

1. public [Date](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\util\Date.html) parse([String](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\lang\String.html) source)throws [ParseException](file:///D:\softwares\Java%20Api%27s\java%20SE\api\java\text\ParseException.html)

* To convert the date from locale specific form to java form.

**Requirement:** Write a program to represent current system date in all possible styles of us format.

**Program:**

import java.text.\*;

import java.util.\*;

public class DateFormatDemo

{

public static void main(String args[]){

System.out.println("full form is :"+DateFormat.getDateInstance(0).format(new Date()));

System.out.println("long form is :"+DateFormat.getDateInstance(1).format(new Date()));

System.out.println("medium form is :"+DateFormat.getDateInstance(2).format(new Date()));

System.out.println("short form is :"+DateFormat.getDateInstance(3).format(new Date()));

}

}

**Output:**

Full form is: Wednesday, July 20, 2011

Long form is: July 20, 2011

Medium form is: Jul 20, 2011

Short form is: 7/20/11

**Note:** The default style is medium style.

**Requirement:** Write a program to represent current system date in UK, US and ITALY styles.

**Program:**

import java.text.\*;

import java.util.\*;

public class DateFormatDemo

{

public static void main(String args[]){

DateFormat UK=DateFormat.getDateInstance(0,Locale.UK);

DateFormat US=DateFormat.getDateInstance(0,Locale.US);

DateFormat ITALY=DateFormat.getDateInstance(0,Locale.ITALY);

System.out.println("UK style is :"+UK.format(new Date()));

System.out.println("US style is :"+US.format(new Date()));

System.out.println("ITALY style is :"+ITALY.format(new Date()));

}

}

**Output:**

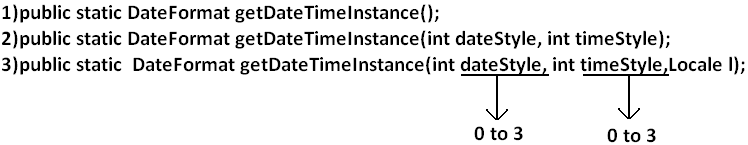
UK style is: Wednesday, 20 July 2011

US style is: Wednesday, July 20, 2011

ITALY style is: mercoled∞ 20 luglio 2011

**Getting DateFormat object to get both date and time:**

* DateFormat class defines the following methods for this.



**Example:**

import java.text.\*;

import java.util.\*;

public class DateFormatDemo

{

public static void main(String args[]){

DateFormat ITALY=DateFormat.getDateTimeInstance(0,0,Locale.ITALY);

System.out.println("ITALY style is:"+ITALY.format(new Date()));

}

}

**Output:**

ITALY style is: mercoled∞ 20 luglio 2011 23.21.30 IST

**SERIALIZATION**

1. Introduction.

2. Object graph in serialization.

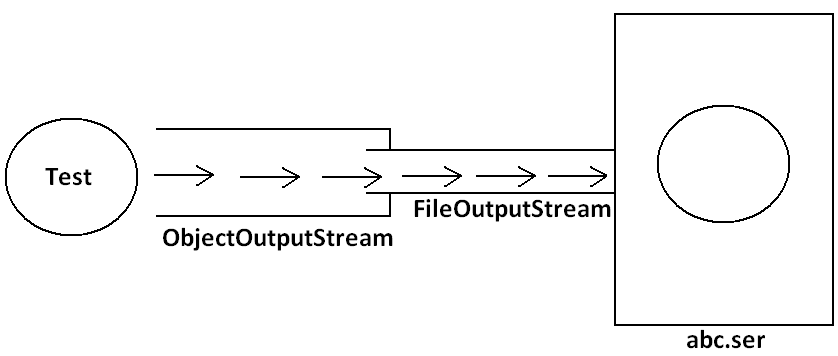
3. customized serialization.

4. Serialization with respect inheritance.

**Serialization:** The process of saving (or) writing state of an object to a file is called serialization but strictly speaking it is the process of converting an object from java supported form to either network supported form (or) file supported form.

* By using FileOutputStream and ObjectOutputStream classes we can achieve serialization process.

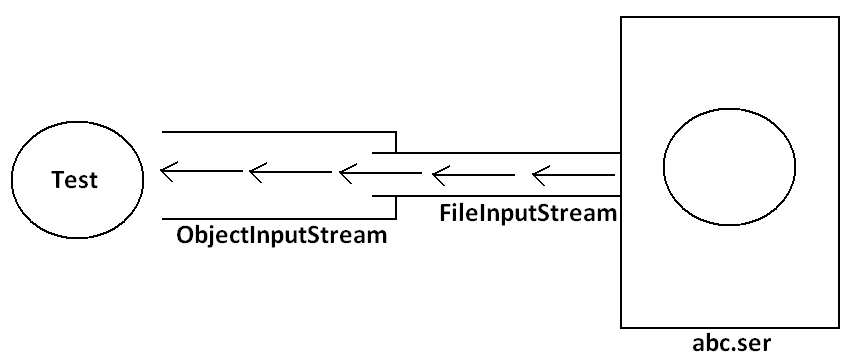
**Diagram:**



**DeSerialization:** The process of reading state of an object from a file is called DeSerialization but strictly speaking it is the process of converting an object from file supported form (or) network supported form to java supported form.

* By using FileInputStream and ObjectInputStream classes we can achieve DeSerialization.

**Diagram:**

****

**Example 1:**

import java.io.\*;

class Dog implements Serializable

{

int i=10;

int j=20;

}

class SerializableDemo

{

public static void main(String args[])throws Exception{

Dog d1=new Dog();

System.out.println("Serialization started");

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(d1);

System.out.println("Serialization ended");

System.out.println("Deserialization started");

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Dog d2=(Dog)ois.readObject();

System.out.println("Deserialization ended");

System.out.println(d2.i+"................"+d2.j);

}

}

**Output:**

Serialization started

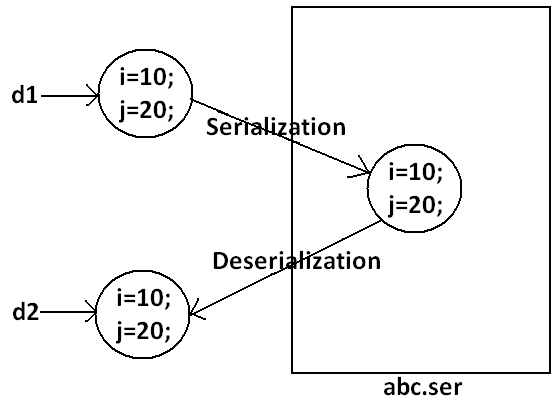
Serialization ended

Deserialization started

Deserialization ended

10................20

**Diagram:**



**Note:** We can perform Serialization only for Serilizable objects.

* An object is said to be Serilizable if and only if the corresponding class implements Serializable interface.
* Serializable interface present in **java.io package** and does not contain any methods. It is marker interface. The required ability will be provided automatically by JVM.
* We can add any no. Of objects to the file and we can read all those objects from the file but in which order we wrote objects in the same order only the objects will come back. That is order is important.

**Example2:**

import java.io.\*;

class Dog implements Serializable

{

int i=10;

int j=20;

}

class Cat implements Serializable

{

int i=30;

int j=40;

}

class SerializableDemo

{

public static void main(String args[])throws Exception{

Dog d1=new Dog();

Cat c1=new Cat();

System.out.println("Serialization started");

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(d1);

oos.writeObject(c1);

System.out.println("Serialization ended");

System.out.println("Deserialization started");

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Dog d2=(Dog)ois.readObject();

Cat c2=(Cat)ois.readObject();

System.out.println("Deserialization ended");

System.out.println(d2.i+"................"+d2.j);

System.out.println(c2.i+"................"+c2.j);

}

}

**Output:**

Serialization started

Serialization ended

Deserialization started

Deserialization ended

10................20

30................40

**Transient keyword:**

* While performing serialization if we don’t want to serialize the value of a particular variable then we should declare that variable with “transient” keyword.
* At the time of serialization JVM ignores the original value of transient variable and save default value.
* That is transient means “not to serialize”.

**Static Vs Transient:**

* static variable is not part of object state hence they won’t participate in serialization because of this declaring a static variable as transient these is no use.

**Transient Vs Final:**

* final variables will be participated into serialization directly by their values. Hence declaring a final variable as transient there is no use.

**Example 3:**

import java.io.\*;

class Dog implements Serializable

{

static transient int i=10;

final transient int j=20;

}

class SerializableDemo

{

public static void main(String args[])throws Exception{

Dog d1=new Dog();

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(d1);

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Dog d2=(Dog)ois.readObject();

System.out.println(d2.i+"................"+d2.j);

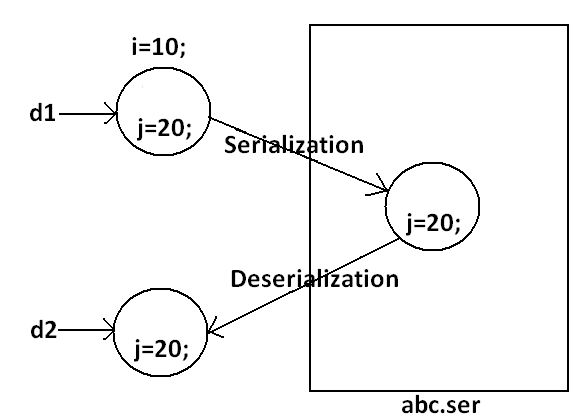
}

}

**Output:**

10................20

**Diagram:**

****

**Table:**

|  |  |
| --- | --- |
| declaration | output |
| int i=10;  int j=20; | 10................20 |
| transient int i=10;  int j=20; | 0................20 |
| transient int i=10;  transient static int j=20; | 0................20 |
| transient final int i=10;  transient int j=20; | 10................0 |
| transient final int i=10;  transient static int j=20; | 10................20 |

**Object graph in serialization:**

* Whenever we are serializing an object the set of all objects which are reachable from that object will be serialized automatically. This group of objects is nothing but object graph in serialization.
* In object graph every object should be Serializable otherwise we will get runtime exception saying “NotSerializableException”.

**Example 4:**

import java.io.\*;

class Dog implements Serializable

{

Cat c=new Cat();

}

class Cat implements Serializable

{

Rat r=new Rat();

}

class Rat implements Serializable

{

int j=20;

}

class SerializableDemo

{

public static void main(String args[])throws Exception{

Dog d1=new Dog();

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(d1);

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Dog d2=(Dog)ois.readObject();

System.out.println(d2.c.r.j);

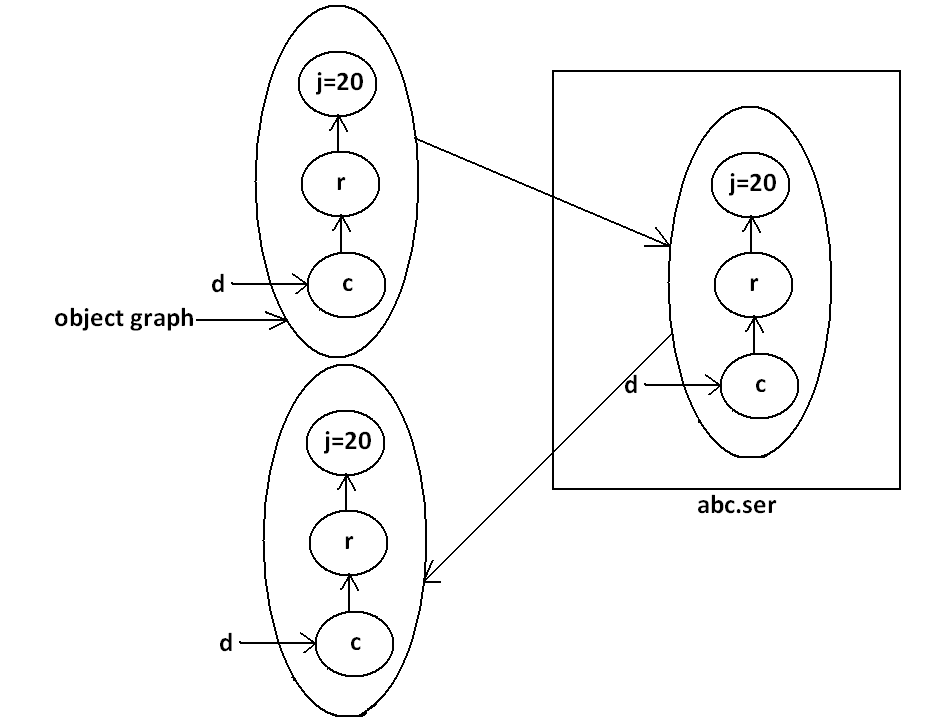
}

}

**Output:**

20

**Diagram:**



* In the above example whenever we are serializing Dog object automatically Cat and Rat objects will be serialized because these are part of object graph of Dog object.
* Among Dog, Cat, Rat if at least one object is not serializable then we will get runtime exception saying “NotSerializableException”.

**Customized serialization:**

**Example 5:**

import java.io.\*;

class Account implements Serializable

{

String userName="Bhaskar";

transient String pwd="kajal";

}

class CustomizedSerializeDemo

{

public static void main(String[] args)throws Exception{

Account a1=new Account();

System.out.println(a1.userName+"........."+a1.pwd);

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(a1);

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Account a2=(Account)ois.readObject();

System.out.println(a2.userName+"........."+a2.pwd);

}

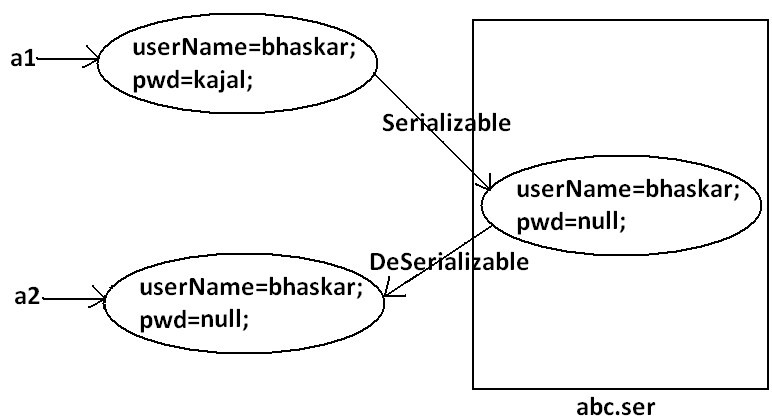
}

**Output:**

Bhaskar.........kajal

Bhaskar.........null

**Diagram:**



* In the above example before serialization Account object can provide proper username and password. But after Deserialization Account object can provide only username bur not password. This is due to declaring password as transient. Hence doing default serialization there may be a chance of loss of information due to transient keyword.
* We can recover this loss of information by using customized serialization.
* We can implements customized serialization by using the following two methods.

1. private void writeObject(OutputStream os) throws Exception.

* This method will be executed automatically by jvm at the time of serialization.
* It is a callback method. Hence at the time of serialization if we want to perform any extra work we have to define that in this method only.

1. private void readObject(InputStream is)throws Exception.

* This method will be executed automatically by JVM at the time of Deserialization. Hence at the time of Deserialization if we want to perform any extra activity we have to define that in this method only.

**Example 6: Demo program for customized serialization to recover loss of information which is happen due to transient keyword.**

import java.io.\*;

class Account implements Serializable

{

String userName="Bhaskar";

transient String pwd="kajal";

private void writeObject(ObjectOutputStream os)throws Exception

{

os.defaultWriteObject();

String epwd="123"+pwd;

os.writeObject(epwd);

}

private void readObject(ObjectInputStream is)throws Exception{

is.defaultReadObject();

String epwd=(String)is.readObject();

pwd=epwd.substring(3);

}

}

class CustomizedSerializeDemo

{

public static void main(String[] args)throws Exception{

Account a1=new Account();

System.out.println(a1.userName+"........."+a1.pwd);

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(a1);

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Account a2=(Account)ois.readObject();

System.out.println(a2.userName+"........."+a2.pwd);

}

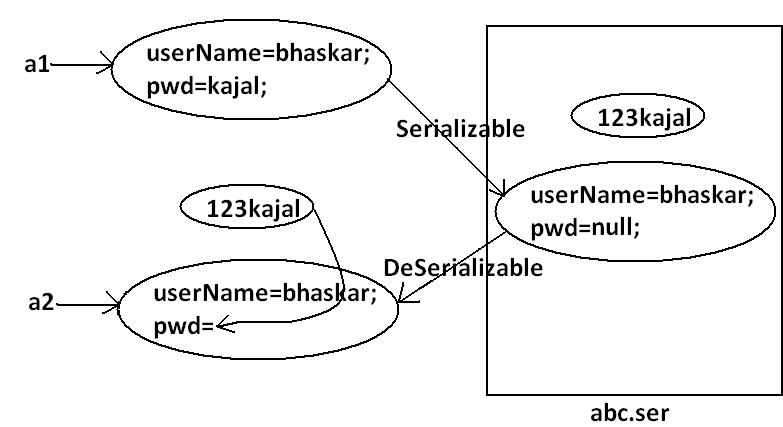
}

**Output:**

Bhaskar.........kajal

Bhaskar.........kajal

**Diagram:**



* At the time of Account object serialization JVM will check is there any writeObject() method in Account class or not. If it is not available then JVM is responsible to perform serialization(default serialization). If Account class contains writeObject() method then JVM feels very happy and executes that Account class writeObject() method. The same rule is applicable for readObject() method also.

**Serialization with respect to inheritance:**

**Case 1:**

* If parent class implements Serializable then automatically every child class by default implements Serializable. That is Serializable nature is inheriting from parent to child.

**Example 7:**

import java.io.\*;

class Animal implements Serializable

{

int i=10;

}

class Dog extends Animal

{

int j=20;

}

class SerializableWRTInheritance

{

public static void main(String[] args)throws Exception{

Dog d1=new Dog();

System.out.println(d1.i+"........"+d1.j);

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(d1);

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Dog d2=(Dog)ois.readObject();

System.out.println(d2.i+"........"+d2.j);

}

}

Output:

10........20

10........20

* Even though Dog class does not implements Serializable interface explicitly but we can Serialize Dog object because its parent class animal already implements Serializable interface.

**Case 2:**

* Even though parent class does not implements Serializable we can serialize child object if child class implements Serializable interface.
* At the time of serialization JVM ignores the values of instance variables which are coming from non Serializable parent JVM saves default values for those variables.
* At the time of Deserialization JVM checks whether any parent class is non Serializable or not. If any parent class is non Serializable JVM creates a separate object for every non Serializable parent and shares its instance variables to the current object.
* For this JVM always calls no arg constructor(default constructor) of that non Serializable parent hence every non Serializable parent should compulsory contain no arg constructor otherwise we will get runtime exception.

**Example 8:**

import java.io.\*;

class Animal

{

int i=10;

Animal(){

System.out.println("Animal constructor called");

}

}

class Dog extends Animal implements Serializable

{

int j=20;

Dog(){

System.out.println("Dog constructor called");

}

}

class SerializableWRTInheritance

{

public static void main(String[] args)throws Exception{

Dog d1=new Dog();

d1.i=888;

d1.j=999;

FileOutputStream fos=new FileOutputStream("abc.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(d1);

System.out.println("Deserialization started");

FileInputStream fis=new FileInputStream("abc.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Dog d2=(Dog)ois.readObject();

System.out.println(d2.i+"........."+d2.j);

}

}

**Output:**

Animal constructor called

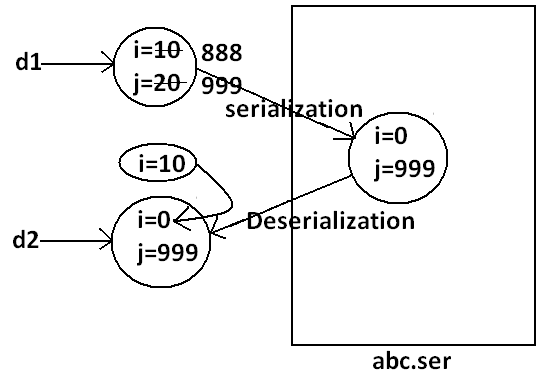
Dog constructor called

Deserialization started

Animal constructor called

10.........999

**Diagram:**

****