

CORE JAVA

By

NAGARJUNA GADDAM

COREJAVA BY NAGARJUNA

Keywords and Reservwrods

Identifier

Java Programming Format (or) Structure

Importance of main method

History of Java Tech

JAVA Versions

DataTypes

JVM (Java Virtual Machine) Archistucture

Variables

COREJAVA BY NAGARJUNA

Key words and Reserve words in java: (53)

-50 keywords and 3 reserve words (true, false, null)

-50 keywords 48 used and 2 unused (goto, constant).

Keywords for Modifiers: (11)

Public, Private, Protected, Strictfp, Static, Abstract, Final, Native, Synchronized, Transient, volatile.

Keywords for Control flow statements: (11)

If, if else, switch, case, break, continue, return, default, for, while, do.

Keyword for class related: (6)

Class, interface, import, package, extends, implements.

Keyword for exception handling: (6)

Try, catch, finally, throw, throws, assert (1.4).

Keyword for Object related: (4)

New, instance of, this, super.

Keyword for return type: (1)

Void.

Keyword to define the group named constants: (1)

Enum (5.0).

Primitive data types: (8)

Byte, int, short, long, float, double, char, boolean.

Java naming convention:

Class

Interface

Method

Variable

Constant

Package

Identifier:

Identifier is a name that is class name, method name, variable name, interface name and ...

Rules for identifier names:

- 1) Identifier name can allowed only the following symbols Alphabets(A-Z, a-z), digits(0-9), special symbols (\$, _ (under square)).

Exp:-

```
Class Test
{
}
```

Exp:-

```
Class Test
{
    Int emp_age;
    Int java@name;//in valid
    Int sal#;// in valid
    Int $Test;
    Int _name;
}
```

- 2) Identifier name should not start with digit, identifier name must be start with a letter(alphabet), a currency character(\$), a connecting character(_).

Exp3:-

```
Class $Test
{
}
```

Exp4:-

```
Class _Test
{
}
```

Exp5:-

```
Class Test
{
}
```

Exp6:-

```
Class OTest
{
    // invalid can't start with digit
}
```

- 3) Key words and reserve words may not be used as an identifier.

Exp7:-

```
class Test
{
    public static void main(String [] args)
    {
        int x=10;
        int assert=20;
    }
}
```

- 4) There is no length limit to the java identifier name but it is suggestible to maintain less than 15 characters to improve the program readability purpose.

Java Programming Format (or) Structure:

To design any java application we must follow java programming format or structure.

Java programming format or structure contains the following sections.

- a) Comment section
- b) Package section
- c) Import section
- d) Class/interface section
- e) Main section

Comment section:

Comments are used to provide the description (Meta data) about our programming implementation.

Comments are optional in java application development.

We can write comments in anywhere in program.

Comments are non-executable part of the java program.

Java tech has supported three types' op comments

- a) Single line comments
- b) Multiline comments
- c) Documentation comment

a) Single line comment:

These comments are used to provide the description with in a single line.

Exp:-

```
// this is my first java program  
Class Test  
{  
}
```

b) Multiline comments:

These comments are used to provide the description in more than one line.

Exp:- /* Java program */

c) Documentation comments:

These comments are used to provide the description in more than one page.

To prepare the documentation comments sun micro system has provided a tool that is "Javadoc".

Package section:

Package is collection of classes and interfaces are grouped into a single unit are called as package. The main advantage of package is to improve the security, shareability and abstraction.

In java every package follows encapsulation mechanism java tech has supported two types of packages.

- 1) Predefined packages
- 2) User defined packages

1) Predefined packages:

These predefined packages are available along with the java software.

- Java.io
- Java.lang
- Java.awt
- Java.Applet
- Java.net
- Java.sql
- Java.util

2) User defined packages:

To define the set of classes and interfaces into a particular package we should use the package declaration statement.

To create user defined package we should use the following syntax

Syn:-

Package Packagename;

```
Package pack1; //valid but not recommend to use
```

```
Package com.icici.loan.homeloan;
```

```
Package com.edu.vista;
```

The package should be unique and should not be duplicated to maintain the unique package names sun micro system has given a convention to developer that is start the user defined package name with the internet domain name in reverse.

Execution of user defined packages:

```
java -d . filename.java (here(.) means current directory).
```

```
Java -d d:\nag\arjun filename.java (here location is specified).
```

We must be use package declaration statement as a first statement in java source files otherwise we will get the compile time error.

Error: Excepted class, interface and enum.

Exp8:-

```
Package pack1;  
class Test  
{  
}
```

Exp9:-

```
Package pack1;  
Package pack2;  
class Demo  
{  
    // invalid  
}
```

Note: due to above reason in single java source file can allowed at most (0 or 1) package statement .

Import section:

To get the set of classes and interfaces from a particular package to present java program we should use the import statement.

Java tech has supported two types of import statements.

- a) Implicit import statement
- b) Explicit import statement

Implicit import statement:

```
Import java.io.*;  
Import java.util.*;  
Import java.sql.*;
```

Explicit import statement:

```
Import java.io.BufferedReader;  
Import java.util.Date;  
Import java.util.ArrayList;  
Import java.sql.Connection;  
Import java.util.HashSet;
```

Note: In java application it is highly recommended to use explicit import statement.

Classes/Interface section:

In general as part of the java application development we are able to provide many more number of classes and interfaces as for the application requirements.

Note:

To save the java file we should use the following conditions.

- 1) If java file is having any public class (or) any public abstract class (or) public interface (or) public enum then we must save that java file with public class name (or) public abstract class name (or) Public interface name (or) public enum name. Otherwise we will get the compile time error.
- 2) If we are not providing any public element in our java application then it is possible to save the java file with main class name.

- 3) Due to the above reason it is possible to use at most (one (or) zero) one public element within a single java file.

Exp10:-

```
Public class A  
{  
}  
  
class B  
{  
}  
  
class C  
{  
}  
  
//valid
```

Exp11:-

```
Public class A  
{  
}  
  
Public class B  
{  
}
```

// in valid a java program can allowed only one public. But it can allowed more the public for variables and methods

Exp12:-

```
Public class A
```

```
{  
Public int Sid;  
Public String sname;  
Public void m1()  
{  
}  
Public void m2()  
{  
}  
}
```

Main section:

It is mandatory section in every java application to execute any java application.

To define the main method in any class that class is treated as main class and it mandatory.

Exp13:-

```
class A  
{  
}  
class B  
{  
Public static void main(String[] args)  
{  
    System.out.println ("hi");  
}
```

```
class C  
{  
}  
  
// We need save class name is B  
  
D:>Java A  
  
//java.lang.NoSuchMethodError: main  
  
D:>java C  
  
//java.lang.NoSuchMethodError: main
```

Note: Weather class contains main method or not and it is properly declared or not these are not responsibility of compiler.

At runtime JVM is responsible for this

At runtime if JVM will not find the main method then we will get the runtime exception like.

Error: Java.lang.NoSuchMethodError: main

The .class file generation is depends on number of classes and interfaces ,number of inner classes and number of abstract classes present in our java program.

Importance of main method:

Public static void main (String [] args)

To execute the java application JVM must access the main() method.

JVM always searches the main method signature as public static void main(String[] args)

Q) Why we should declare main() method as public?

a) public is access modifier it can be used to define the scope for java programming constructs like method, variable, interface and

b) When we installed java s/w on our machine automatically JVM will be installed in the form of java tools in “bin” directory of JDK s/w.

- c) If u design any java application the main method is available in our application directory structure.
- d) For making availability of main() method scope to JVM we must declare main() as public.
- e) If we are not define main() method as public then compiler will not raise any error.

But JVM will raise an error message like.

Error: Main() method not public

Note: From java7.0 version on words we will get the following error. Main method not found in class XXX please define the main method as public static void main (String[] args).

Exp:-

```
class Test
{
    Static void main(String[] args)
    {
        System.out.println("hello");
    }
}
```

// above we will get the runtime error.

Q) Why we should declare main method as static?

They are two types of methods.

- a) Static
- b) Non-static(instance)

Static method:

Static method could be accessed by either using the respective class objective reference (or) respective class name directly.

Within the same class we can access the static methods directly.

Exp:-

```
class Test3
{
    public static void m1()
    {
        System.out.println("m1()-method");
    }
    public static void m2()
    {
        System.out.println("m2()-method");
    }

    public static void main(String[] args)
    {
        System.out.println("main()-method");

        m1();
        m2();

        Test3 t=new Test3();
        t.m1();
        t.m2();

        Test3.m1();
        Test3.m2();
    }
}
```

D:\Nagarjuna_java\JavaExp>javac Test3.java

D:\Nagarjuna_java\JavaExp>java Test3
main()-method
m1()-method
m2()-method
m1()-method
m2()-method

m1()-method
m2()-method

Non- Static Method:

We can access the non-static methods directly from non-static area(non-static methods, non-static blocks).but if we want to any non-static methods from the static area (static –block, static –methods) we must be use object reference otherwise we will get compile time error.

Error: Non-static method XXX cannot be referenced from static area

As per the predefine implementation of JVM, JVM will access the main method with the provided class name along with the java command.

To access the main method with class name by JVM we must declare main method as static.

If we are not declare main method as static then compiler will not raise any error but JVM will raise an exception.

Error: Java.lang.NoSuchMethodError: main

From java 7.0 version on words we will get the following exception.

Error: Main method non-static XXX class XXX please define main() public static void main(String[] args)

Exp:-

```
Class Demo
{
    public void main (String[] args)
    {
        System.out.println ("Hello World!");
    }
}
```

Along with public and static modifiers the main method is allowed the following modifiers.

Final
Synchronized
Strictfp

Verify the following main method declaration

- 1) Public static void main(String[] args)
- 2) Static public void main(String[] args)
- 3) Public void main(String[] args)
- 4) Static void main (string[] args)
- 5) public Synchronized void main(String[] args)
- 6) public Strictfp void main(String[] args)
- 7) public static abstract void main(String[] args)(public and abstract combination is always illegal)
- 8) Public static final Strictfp void main(String[] args)

Void:

Q) Why we provide void has return type to main?

If the method return type is void it is called void method, else it is called non-void type method.

Non-void type method must be return a value after its execution and also that value must be same as Method returns type.

Exp:-

```
Public int m1()
{
    Return 10;
}
```

Exp:-

```
Public int m2()
{
    Return 10.1;
}
// compile time error must be return same integer value
```

Exp:-

```
Public int m3()
{
}
// compile time error is missing return statement.
```

If we don't place return statement in non-void type method then compiler will raise an error like
" missing return statement".

Return Statement: The return statement is used to terminate the method execution and per sending control back to calling method.

Return: it is allowed only in void type methods and constructors and it is optional.
Return value: it is allowed in only non-void type methods and it is mandatory.

Void type: if we don't want to return anything from a method then we will declare that method type has void.

Main method won't be return anything to JVM so we can declare main method as void return type.

If we are not provide void has a return type to the main method then compiler will not raise any error, but JVM will raise an exception like

Error: Java.lang.NoSuchMethodError: main

Main: To show up the power of that main method in java tech has given name to this method has "main". And it is mandatory.

Parameters to main (String [] args):

The main purpose of parameters to the main method to accept the command line arguments.

Command line Arguments:

It is input values are passed along with the java command, On the command prompt.

Args:

Args is a variable name in the place of args we can use any java valid identifier name then it is valid.

Exp:-

```
class Demo1
{
    Public static void main(String[] Nag)
```

```
        {
            System.out.println (Nag[0]);
        }
    }
```

History of Java Tech:

The language java developed by sun micro system (now a day it hands over by Oracle Corporation). In the year of 1991 by a person called James Gosling and its team.

Sun micro System is originally an academic university and developed rules for developing java and those rules are programmatically developed by the team of members.

In the year of 1991 they are released a trial version name is OKA (it is a tree name). OKA is known as original name of Java and it has taken 18 month to develop.

But the s/w OKA is not fully solving the industry oriented problems.

Sun micro system has adding some additional features and released on the name of java in the year of 1995 java means it is seed name.

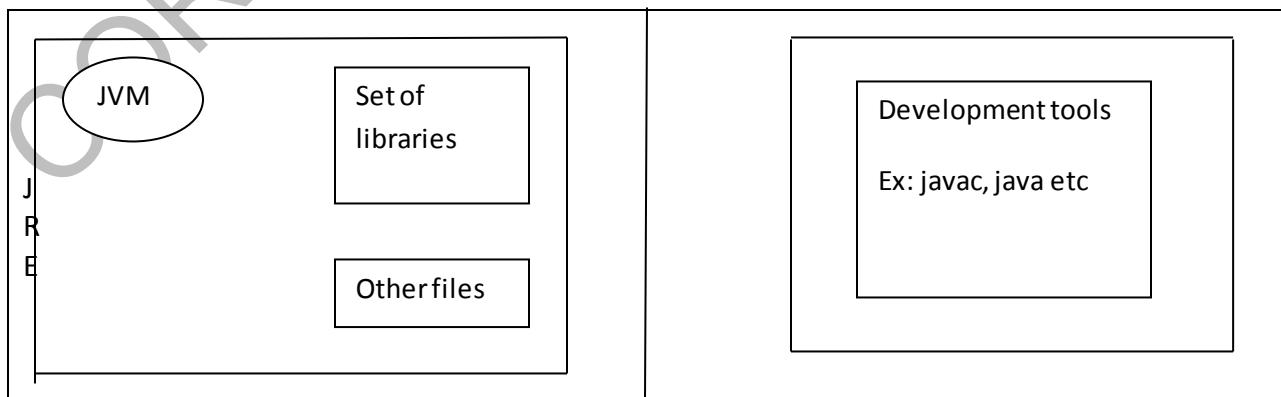
Types of Java Software's:

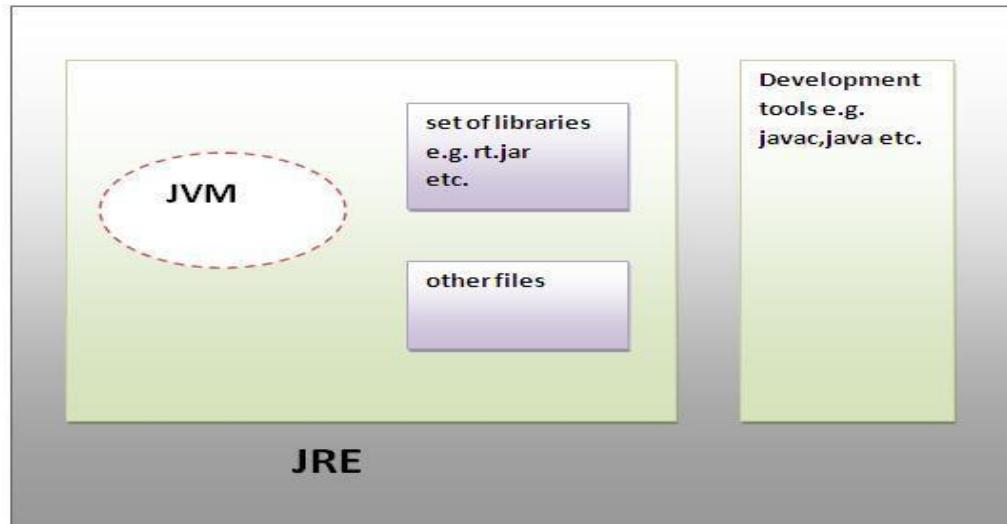
Java s/w is divided into two parts

JDK (java development kit).

JRE (java Runtime Environment).

JDK: JDK has both compiler and JVM so using JDK we can develop compile and execute the new java applications and also we can modify, compile and execute already developed application.

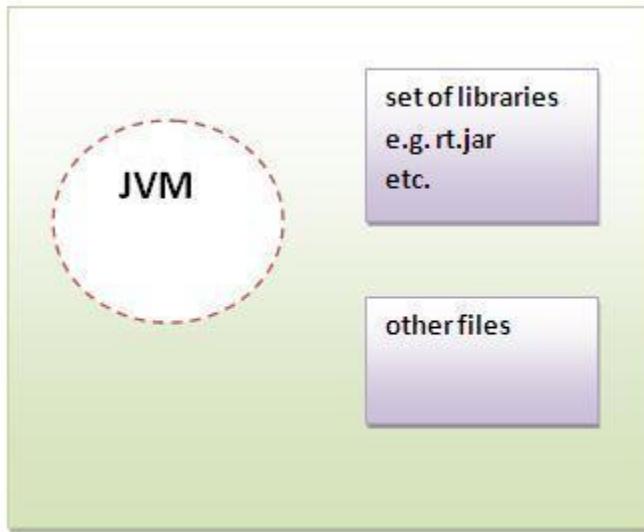




JDK

JRE:

JRE has contains only JVM hence using JRE we can execute only already developed applications.



JRE

JVM(Java Virtual Machine):

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java byte code can be executed.

JVMs are available for many hardware and software platforms (i.e.JVM is platform dependent).

The JVM performs four main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

Note:

JVM is subset of JRE, JRE is subset of JDK.

Installation of JDK and JRE:

Run the downloader installer it will install both JDK and JRE.

Note:

JDK installed directory is called java home directory.

Setting the path environment variables:

The java s/w has below two files

Binary files

Library files

Binary files are command files by default binary files are stored in a folder call “bin”.

Library files are program files internally they have logic that is used to develop another application; library files are stored in folder call “lib”.

Java programs are compiled and executed by using binary files like javac and java.

We can call this a binary file as tools in short form.

After java s/w installation we should make available all the JDK tools to operating system. Otherwise if we are using that tool then we will get following error message.

Error: Javac is not a recognized internal or external or operable or batch file.

To achieve this to set path environment variable to the location, where all the JDK tools are available that is bin folder of JDK home directory.

Path environment variables settings are finding the binary files.

Class path environment variables settings are for finding the library files.

Path environment variables:

We have two ways to set the path environment variables.

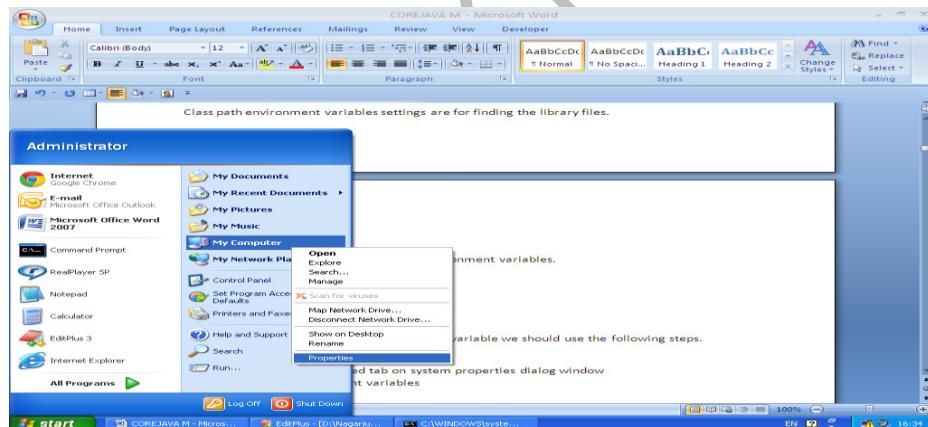
Permanent setting

Temporary setting

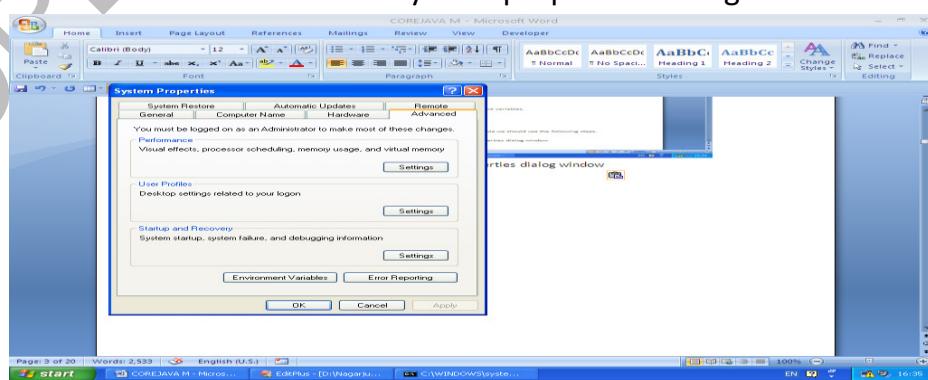
Permanent setting:

To set Permanent path environment variable we should use the following steps.

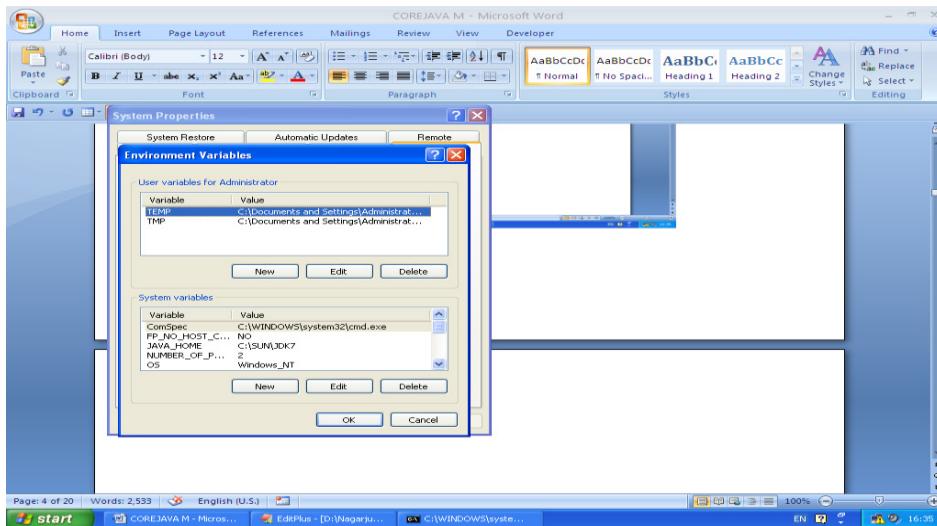
Click on my computer



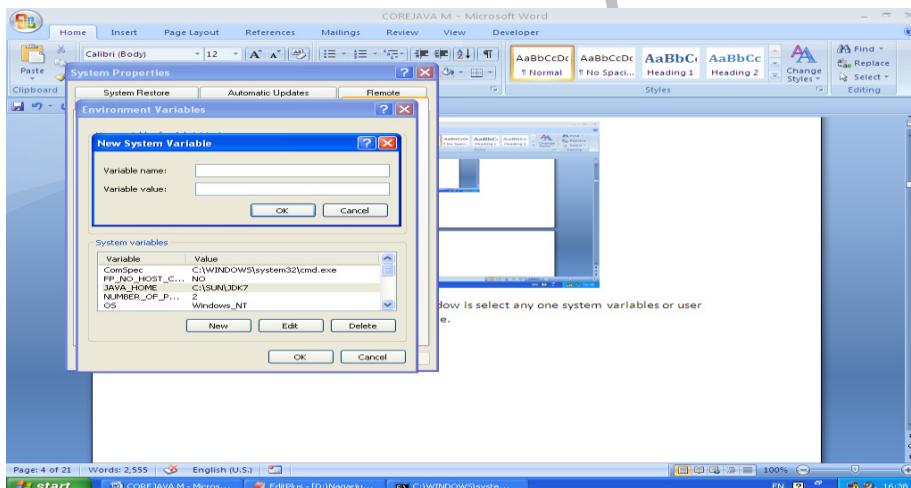
Select the advanced tab on system properties dialog window



Select environment variables



Then a new environment variable dialog window is select any one system variables or user variables to set the path environment variable. click on new button it display new system variable window is appear.



In above enter the variable name: we can write path (or) JAVA_HOME

Variable values: we can type C:\SUN\JDK7\bin

Then click on OK button three times.

Temporary setting: To set the environment variables temporarily we should use the command prompt because those settings are available only for that command prompt till it is closed. Once we close that command prompt all the settings are gone.

Setting Java binary files setting:

Set path=%path%;c:\Java\JDK7\bin;

Setting Java library files setting:

Set class path=%class path%;.c:\Java\JDK7\lib;

JAVA Versions:

Java Name	Code Name	Release Date	Concept
JDK1.0	OAK	Jan23, 1996	All Versions
JDK1.1	OAK	Feb19, 1997	Introduced Awt event model, Inner classes, java bean, JDBC
J2SE1.2	Play ground	Dec8, 1998	Introduced J2EE, J2ME, JFC (java Foundation classes, collection Frame work and JIT compiler.
J2SE1.3	Kestrel	May8, 2000	Introduced Hot Stop, JVM
J2SE1.4	Merlin	Feb6, 2002	assert, non-blocking IO, logging API Java web start, regular expressions.
JSE5.0	Tiger	Sept30, 2004	generics, auto-boxing, auto un- boxing Annotations, enum, var args, For each loop, static import (fail).
JSE6.0	Mustang	Dec11, 2006	Navigable Map, Navigable Interface In java.util package.
JSE7.0	Dolphin	July28, 2011	try with resource statement, Diamond operator

Data Types:

Data types are used to store the data temporarily in computer through a program.

Def: Data type is something which give information about size of the memory, location and range of the data that can be accommodated inside a particular location.

What type of result come out from an expression when we use these types inside the expression?

Exp:- Int= 10+10; Java tech has supported two types of data types

Primitive data types

Non-primitive (or) referenced data types (or) user-defined data types.

Primitive data types:

Java tech has supported 8 primitive data types that 8 primitive data are divided into two types.

Numerical data types

Non-numerical data types

Primitive data types

Numerical

Integer DT

Byte

Int

Short

long

floating DT

float

double

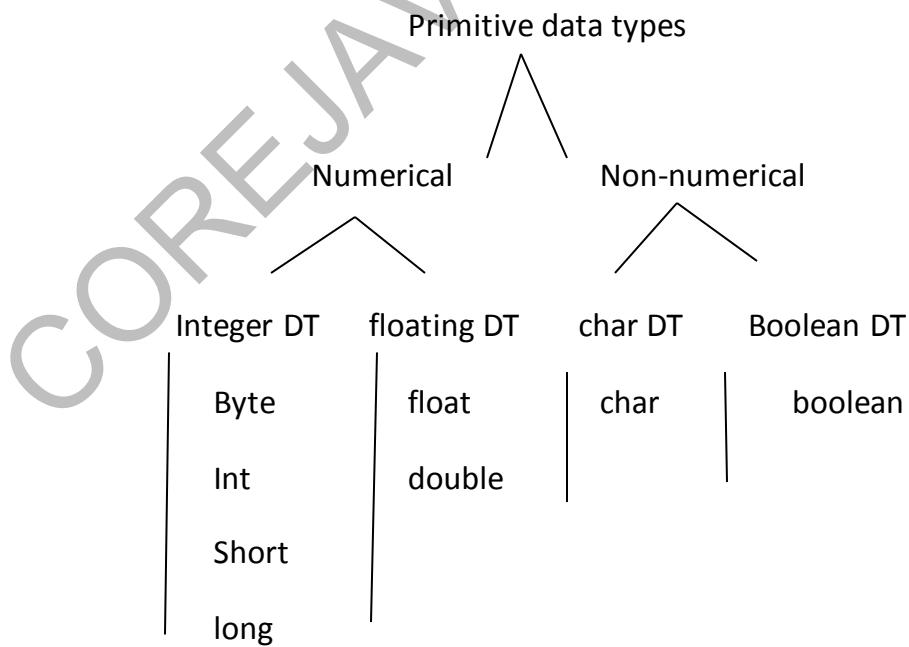
Non-numerical

char DT

char

Boolean DT

boolean



Integer data types:

byte

int

short

long

byte: it allows only integer values other than that it will not allow.

size: 1 byte (or) 8 bits

byte b1=10;

Range : -128 to 127

byte b2=30;

byte b3=b1+b2; error it should the int not byte.

Exp:-

```
class Test2
{
    public static void main(String[] args)
    {
        byte b1=127;
        byte b2=-129;
        byte b3=1.1; //invalid
        byte b4=false; //invalid
        System.out.println(b1);
        System.out.println(b2);
    }
}
```

Error: possible loss of precision

```
byte b2=-129;
```

Whenever we are performing mathematic operations b/w the two primitives we must be maintain result at type has max data type.

Exp:-

byte + byte= int

byte + int= int

byte + short= int

byte + long= long

byte + float= float;

byte + double= double

byte + char= int

short: it allows only integer values other than that it will not allow.

Size : 2 bytes

Range : -32768 to 32767

int: it allows only integer values other than that it will not allow.

Size: 4 bytes

Range : -2147483648 to 2147483647

Range : -3.4 e 28 to 3.4 e 28

long: : it allows only integer values other than that it will not allow.

Size: 8 bytes

Range : -9223372036854775808 to 9223372036854775807

floating data types:

These data types are use full to represent numbers with decimal point. For exp 3.14, 5.0

float:

size: 4 bytes

range: -3.4 e 38 to 3.4 e 38

double:

size: 8 bytes

range: -1.8 e 308 to 1.8 e 308

Difference b/w float and double:

float can represent up to 7 digits accurately after decimal point, whereas double can represent up to 15 digits accurately after decimal point.

Non-numerical data types:

Char

boolean

character data type:

character data type can represent a single character like a, b, p, &, *

char:

size: 2 bytes

range: 0 to 65535

boolean data type:

boolean data type can represent any of the two values true (or) false. JVM uses 1 bit to represent a boolean value internally

Exp:- boolean response=false;

Literals:

A literal represents a value that is stored into a variable directly in the program

Exp:-

```
Char gender='M';  
Int age=25;  
Short s=40000;  
Boolean result= false;
```

Above exp right hand side values are called literals because these values are being stored into the variables shown at left hand side. And the data type of the variable changes, the type of literals also changes. So we have different type literals

- Integer literals
- Float literals
- Character literals
- String literals
- Boolean literals

Integer literals:

Integer literals represent the fixed integer values like 100, -60, 1234450,

Exp:-

```
Int decVal=26;
```

Floating point literals:

Float literals represent fractional numbers. These are numbers with decimal point like 2.0, -1.2, 0.005, 3.15, 1e-22, etc which should be used with float (or) double type variables. While we write the literal we can use E (or) e for scientific notation, Float for F (or) f , and D (or) d for double literal (this is the default and generally omitted).

Exp:-

```
double d=123.5;
```

double d1=1.234e2; this same as above it is scientific notation.

```
Float f = 123.4f;
```

Character literals:

It is allowed by char data type the char literals are enclosed with single quotation (' '),

General character, like A, b, 9 etc.

Special character, like ?, @, etc.

Escape character, like \n, \b etc.

Unique code character, like \u0042 (this represent a in ISO Latin 1 character).

String literals: String literals represent objects of String class, it will enclosed with double quotation ("").

Exp:-

Hello, Nag, AP101, etc.

Boolean literals:

Boolean literals represent only in two values true (or) false

Data type default values:

Data type	Default values
byte	0
int	0
short	0
long	0
float	0.0
double	0.0
char	space
boolean	false
string	null

JVM (Java Virtual Machine) Archistructure:

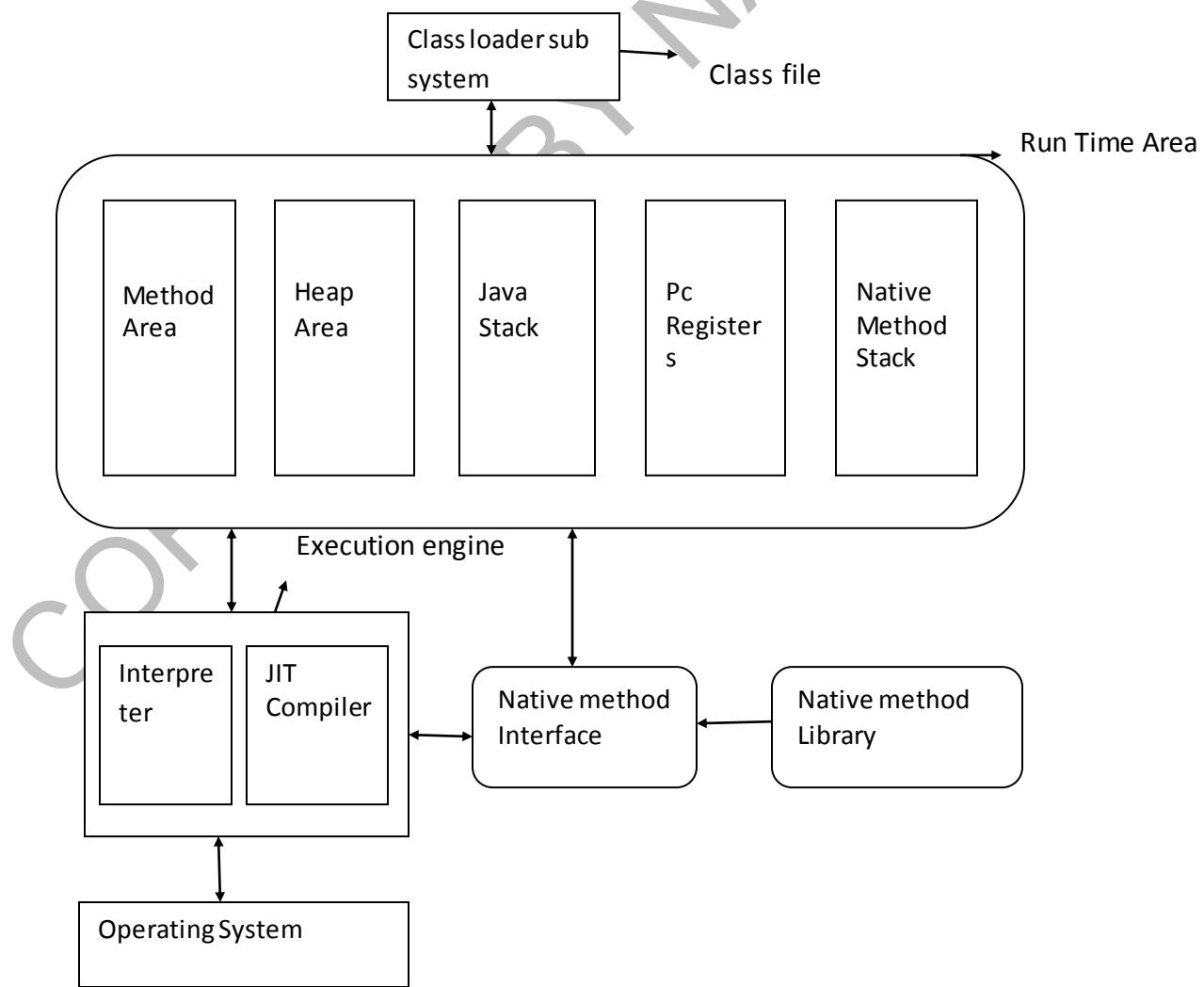
JVM is the heart of entire java program execution process. It is responsible for taking .class file and converting each byte code instruction into the machine language instruction that can be executed by the microprocessor.

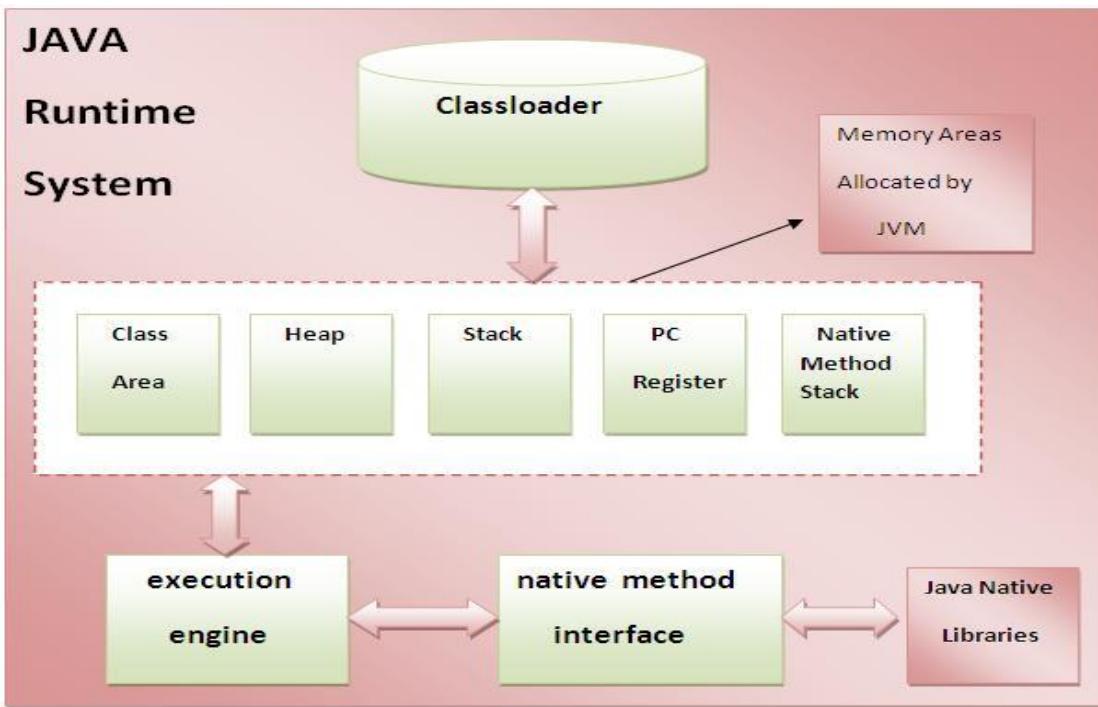
Java byte code:

To implement WORA (Write Once run Anywhere) the JVM uses java byte code, it is middle language in b/w java (user language) and the machine language. JVM can understand only byte code instructions.

.class file:

The .class file itself binary file that can't understand by a human, to manage that file JVM vendors provides javap tool.





The java program is converted into a .class file consisting of byte code instructions by the java compiler. Remember java compiler is outside the JVM, know this .class file is given to the JVM, in the JVM there is a module (program) called class loader sub system, which performs the following instructions.

Class loader sub system:

- a) It loads the .class file into memory.
- b) It verify the byte code instruction whether all byte code instruction are proper (or) not. If it finds any instruction is suspicious, the execution is rejected immediately.
- c) If byte code instructions are proper, then it allocates necessary memory to execute
- d) the program .

This memory is divided into five parts, called Run time data areas, which contains the data and results while running the program. These areas are as follows.

Method Area:

Method area is memory block, which stores the class code, code of the variables, and code of the methods in the java program. (Method means functions written in a class)

Heap Area:

This is the area where the Object is created. Whenever JVM loads a class a method that time heap area are immediately created in it.

Java Stacks:

Method code is stored on method area. But while running a method, it needs some memory area to store the data and results. This memory is allotted on java stacks. So, Java Stacks are memory areas where java methods are executed. While executing methods, a separate frame will be created in the Java Stacks, the method is executed. JVM uses a separate thread (or process) to execute each method.

PC (Program Counter) Registers:

These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instructions of the method.

Native method stack:

Java methods are executed Java Stacks. Similarly, native methods (for exp c/c++ functions) are executed on native method stacks. To execute the native methods, generally native methods libraries (c/c++ header files) are required. These header files are located and connected to JVM by a program, called Native method interface.

Execution engine:

Execution engine contains interpreter and JIT (Just In Time) compiler, which are responsible for converting the byte code instructions into machine code so that the processor execute them. Most of the JVM implementations use both the interpreter and JIT compiler simultaneously to convert the byte code. This technique is also called adaptive optimizer. Generally any language like (C/C++, FORTRAN, COBOL), will use either interpreter (or) JIT compiler to translate the source code into a machine code.

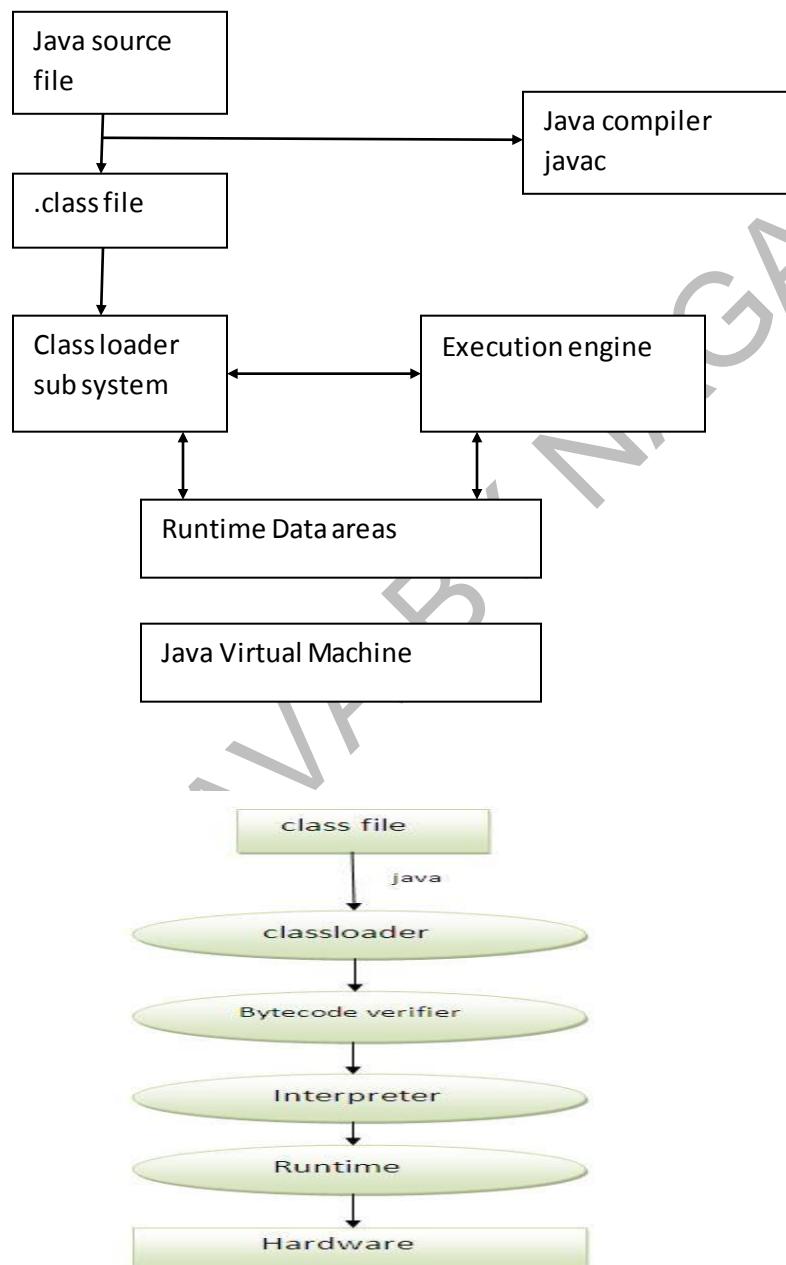
Native method interface:

Native method interface is a program that contains native methods libraries (C/C++ header files) with JVM for executing Native Methods.

Native method library:

It contains native libraries information.

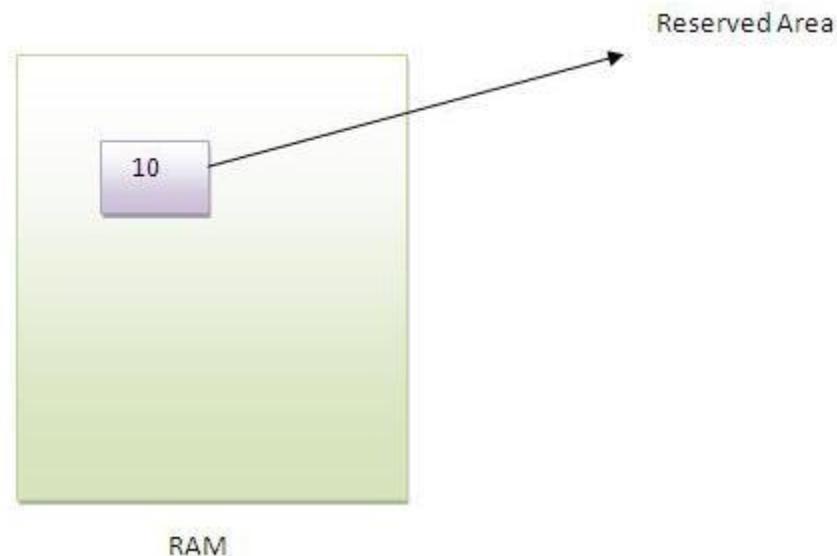
Java Code Execution Process:



Types of variables:

Variable is named memory location based on the position and declaration of the variables. These variables are divided into three types.

- a) Local variables
- b) Static variables
- c) Non-static variables (or) instance variables.



Local variables:

We can declare any variable inside a method (or) inside a block (or) inside a constructor that type of variable is called local variable.

Exp:-

```
public void add()  
{  
    int a=10;  
    int b=30;
```

```
//local variables
```

Exp:-

```
public void add(int a, int b)  
{  
//local variables
```

Exp:-

```
static  
{  
int a=10;  
int b=30;  
//local variables
```

Before using the local variables we must be provide initialization otherwise we will get the compile time error like

Variable XXX might not have been initialized

Exp:-

```
class Test  
{  
public static void main(String[] args)  
{  
int a;  
int b;  
System.out.println (a+b);  
}
```

```
}
```

We are not using the local variables that time no need to provide the initialization to the local variables.

Exp:-

```
class Test  
{  
    public static void main(String[] args)  
    {  
        int a;  
        int b;  
    }  
}
```

The local variable are allowed only final modifier. If we are allowed other then the final modifier then we will get the compile time error.

Exp:- class Test6

```
{  
    public static void main(String[] args)  
    {  
        final int a=40;  
        System.out.println(a);  
    }  
}
```

Exp:- class Test6

```
{
```

```
public static void main(String[] args)
{
    Private int a=40;
    Public int b=38;
    Static int c=39;
    System.out.println(a);
}
```

Error: illegal start of expression

```
Private int a=40;
Public int b=38;
Static int c=39;
```

Note: We can access the local variable with in the method (or) block (or) constructor only.

Instance variables (or) non-static variables:

Instance variables are class level variables we can declare with in class directly.

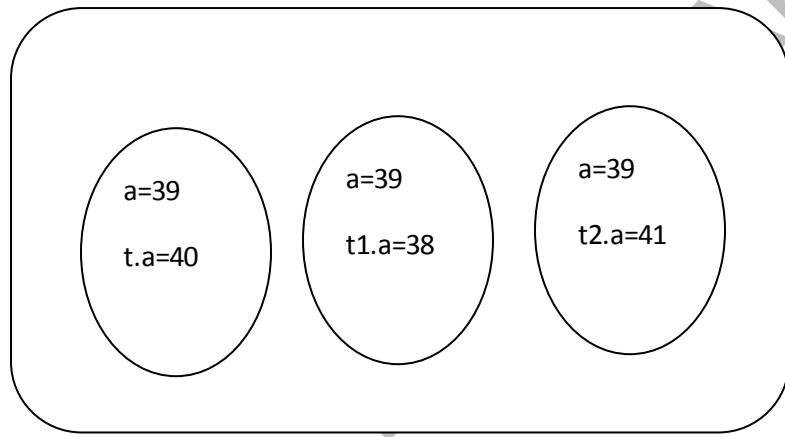
If the value of variable is valid from object to object that type variable is called instance variables (or) non-static variables.

For every object a separate copy instance variable will be created.

Exp:-

```
class Test
{
    int a=39;
    public static void main(String[] args)
    {
        Test t=new Test();
    }
}
```

```
Test t1=new Test();  
Test t2=new Test();  
t.a=40;  
t1.a=38;  
t2.a=41;  
System.out.println(t.a);  
System.out.println(t1.a);  
System.out.println(t2.a);  
}
```



The non-static variables are recognized and executed exactly at the time of object creation.

We can access the non-static variables from non-static area (non-static methods, non-static blocks) directly.

Exp:-

```
class Demo5  
{  
    int a=39;  
    public void m1()  
    {  
        System.out.println(a);  
    }  
}
```

```
}

public static void main(String[] args)
{
    Test t=new Test();
    t.m1();
}

}
```

If we want to access the non-static variables from static area (static methods, static blocks) we must be create the object reference otherwise we will get the compile time error.

Exp:-

```
class Demo5

{
    int a=39;

    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
}
```

JVM will always provide default initialization to the non-static variables.

Exp:-

```
class Demo6

{
    int age;
    String name;
```

```
float sal;

public static void main(String[] args)
{
    Demo6 d=new Demo6();
    System.out.println(d.age);
    System.out.println(d.name);
    System.out.println(d.sal);
}
```

Static variables:

Static variables also class level variables we can declare the static variables inside a class directly, with using static modifier.

The value of variable is not valid from object to object that type of variable is called static variables.

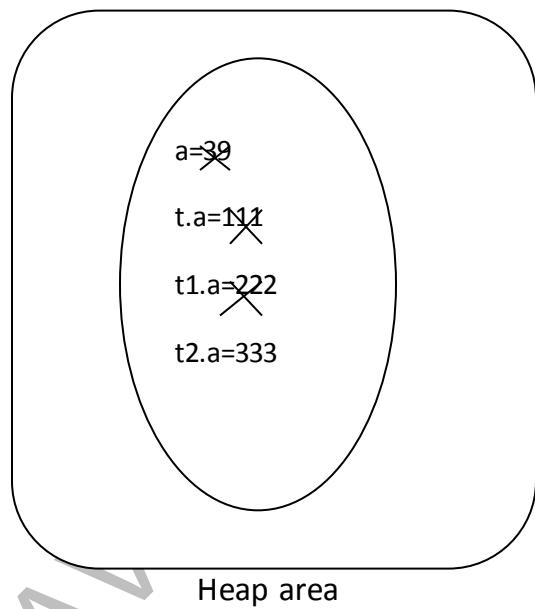
Difference b/w static and non-static variables:

In the case of non-static variable for every object a separate copy will be created. But in the case of static variables for entire class only one copy will created and shared by all the object of that class.

Exp:-

```
class Test9
{
    static int a=39;
    public static void main(String[] args)
    {
        Test9 t=new Test9 ();
        Test9 t1=new Test9 ();
        Test9 t2=new Test9 ();
        t.a=111;
```

```
t1.a=222;  
t2.a=333;  
System.out.println (t.a);  
System.out.println (t1.a);  
System.out.println (t2.a);  
}  
}  
o/p: 333,333,333
```



We can access the static variables from static area with the help of respective “class name” (or) with respective object reference are directly.

Exp:-

```
class Test10  
{  
    static int a=39;  
    public static void main(String[] args)  
    {
```

```
Test10 t=new Test10();

System.out.println(t.a);

System.out.println(Test10.a);

System.out.println(a);

}

}
```

The static variables are recognized and executed exactly at the time of loading respective class byte code into the memory.

JVM always provide the default initialization to the static variables.

```
Exp:- class Demo6

{

    static int age;

    static String name;

    static float sal;

    public static void main(String[] args)

    {

        Demo6 d=new Demo6();

        System.out.println(d.age);

        System.out.println(d.name);

        System.out.println(d.sal);

    }

}
```

Chapter-2: OOP'S

- 1. Class**
- 2. Object**
- 3. Constructor**
- 4. This Keyword**
- 5. Inheritance**
 - Single
 - Multiple
 - Multilevel
 - Hybrid
- 6. Super keyword**
- 7. Abstract class & Methods**
- 8. Interface**
- 9. Polymorphism**
 - Static (Method overloading, Method hiding)
 - Dynamic (Method overriding)
- 10. Encapsulation**
- 11. Abstraction**
- 12. Packages**
- 13. Access Modifiers (public, private, protected, <default>)**
- 14. Type Casting**
- 15. Wrapper classes**

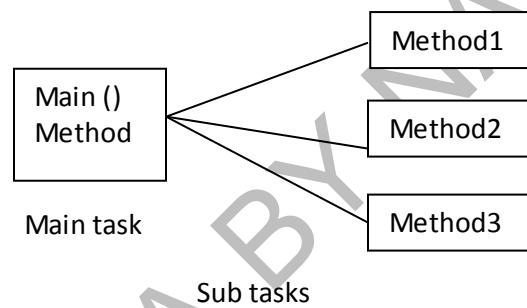
Class :

There are two types of programming models are available. There are

- 1) Procedure oriented programming language.
- 2) Object oriented programming language.

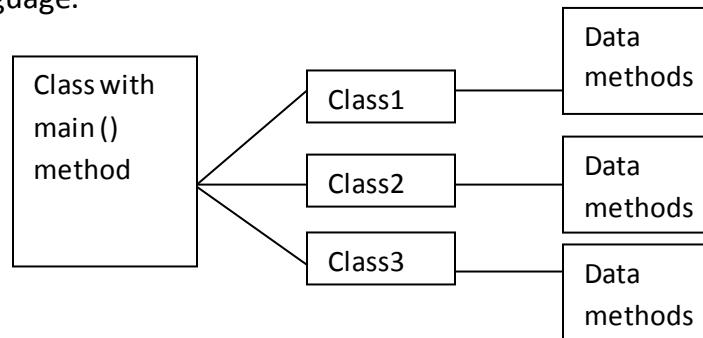
Procedure oriented programming:

The languages like C, Pascal, Fortran, etc., are called procedure oriented programming language, this languages a programmer uses procedure (or) functions to perform a task. A programmer is divided the task into several sub tasks each of which is expressed as a function. So C program generally contains several functions which are controlled from a main () function. If you represent the data by using procedure oriented programming language then there is no security to our data. Procedure oriented programming language provide less security.



Object oriented programming:

The languages like C++, Java uses classes and objects in their programs and we are called object oriented programming languages. A class is model which itself contains data and methods to achieve the task. The main task is divided into several modules, and these are represented as classes. This approach is called object oriented programming language.



If represent the data in object oriented programming language then that language provide more security, so the object oriented programming language provide more security to our data.

Object oriented:

Any technology is called object oriented then we must represent data in the form of objects. And that technology must satisfy the all the object oriented features.

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

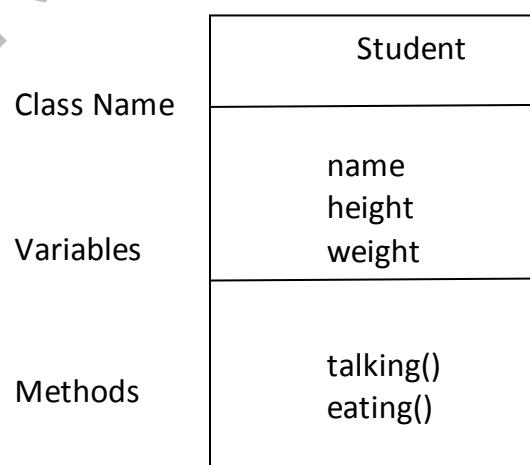
Class:

Class is plan (or) model (or) template (or) blueprint.

Def: A class is way of binding the data Members and associated methods in a single unit.

Any java program if you want to develop then we must and should write with respective class only.(without class there is no java program).

Class is a speciation it can be used to specify the variables and methods. Here variables are called properties (or) attributes and methods are called actions (or) behaviors.



Syntax:

[Modifier] class class name/interface name [extends super class] [implements interface]
{ }

Modifiers:

Modifiers are used for to define scope and some extra nature for java programming element.

Java top level class is allowed only following five modifiers.

Public
<default>
Final
Abstract
Strictfp

Note: The inner class is allowed following 8 modifiers

Along with above five
+
Private
Protected
static

Inner class: declaring a class inside another class is called inner class.

Exp:-

```
Class Test
{
    class Test1
    {
    }
}
```

In the java class syntax it is allowed more than one modifier at a time. The order of modifier is not important. But combination of modifier is very important. (Here abstract and final modifiers combination is always illegal).

The abstract and final combination is always illegal for class and methods.

Exp:-

```
Abstract final class Test // illegal  
{  
}
```

Exp:-

```
Public abstract Strictfp class Test // valid  
{  
}
```

When we declaring a class that time memory is not allocated for the data members of a class. To allocate the memory for those data members of a class we should create an object.

Object:

The instance of a class is called as an object. Instance is nothing but allocating memory for the data members of a class. Object creation represents allocating the memory to store the actual data of the variables. To create an object, we need to follow the below syntax

Syn: Class name object reference=new class name ();

Here 'new' is an operator that creates the object to particular class (which we are using class name for creating the object).

Blue print of a class is nothing but an object

Logical runtime entity of class is nothing but an object

Real world entity of a class is nothing but an object

Object characteristics:

Object contains the mainly three characteristics

- State
- Identity
- Behavior

State:

State describes the data stored in the object that is object properties like

Name, age, height, weight, color etc...

Behavior:

Describes methods in the object, methods is nothing but performing the actions like

Sleep (), talking () , eating (), running ()...

Identity:

JVM produces a unique number for the object from the memory address of the object. This reference number is also called hash code number.

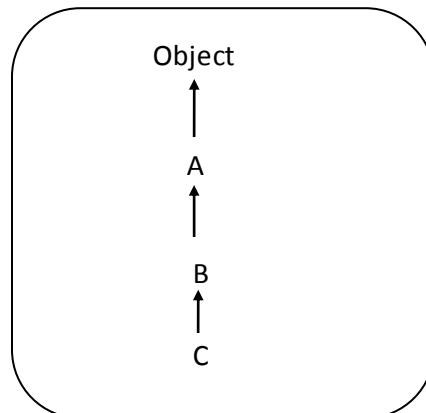
Note: Java object identity is called hash code. Hash code is a 32- bit random integer number created by JVM for every object. For every object we have a unique identity number(hash code)

The hash code of an object can be retrieved by using a method called hashCode ()

Syn:- public native int hashCode ()

hashCode () method present in a Object class

Object class is present in Java.lang package object class is act as a root class in java. Every java class is the child class of object class either directly (or) indirectly.



Object class contains 11 methods this methods are available to all the java classes.

Note: every object contains the hashCode () and object reference.

Syntax:

Classname referencevariable = new classname (parameter list);

Exp:-

```
class Human
{
    String name="nag" ;
    int age=26;
    void talk()
    {
        System.out.println("Hello I am "+ name);
        System.out.println("My age is "+ age);
    }
    public static void main(String[] args)
    {
        Human h= new Human();
        h.talk();
        System.out.println("hash code="+ h.hashCode());
    }
}
```

Note:

- Developer can also generate the hash code based on the state of that object by overriding the hashCode method.
- In this case, if state is changed automatically hashCode () will be change.
- JVM generated hashCode will not be changed if the object state is changed.
- hashCode is used by JVM when the data is stored into the hash table.

Local preference:

When we call a variable compiler and JVM will search for it's definition in method first, if it's definition is not found in method then it will search for class level, if it's definition is not found at class level also then compiler will raise an error like

Syn: can't find symbol

Hence if a variable defined in both method and class level with same name, compiler and JVM will access that variable first method level (first), because always local variable get first priority.

Exp:-

```
class LocalPre
{
    static int a=10;
    static int b=30;
    public static void main(String[] args)
    {
        System.out.println(a);
        int a=40;
        System.out.println (a);
        System.out.println (LocalPre.a);
    }
}
```

Note: If local and static variables contains same name we should use the class name to access the static variables, to differentiate the static and local variables.

Constructor:

Constructor is special member function it is used to provide initialization to instance properties of an object.

Rules:

The constructor name and class name should be same.

Exp:-

```
class Test  
{  
    Test ()  
}  
}
```

The constructor is allowed only four modifiers

- 1) public
- 2) private
- 3) protected
- 4) <default>

If you are using any other modifier then we will get the compile time error

Exp:-

```
class Sample  
{  
    static Sample()  
    {  
    }  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World!");  
    }  
}
```

Error: modifier static not allowed here

static Sample()

Note: If any class contains private constructor then we can create an object with in the same class only. It is not allowed to create an object outside the class.

Exp:-

```
// Creating the object to same class it contains private constructor.

class Sample

{

    private Sample()

    {

    }

    public void m1()

    {

        System.out.println("Hi");

    }

    public static void main(String[] args)

    {

        System.out.println("Hello");

        Sample s=new Sample();

        s.m1();

    }

}
```

Exp:- // Creating the Object outside class the class contains private constructor

// It shows a compile time error.

Error: Sample () has private access in Sample

Sample s=new Sample ();

```
class Sample

{

    private Sample()

    {

    }
```

```
public void m1()
{
    System.out.println("Hi");
}
}

class Sample1
{
    public static void main(String[] args)
    {
        System.out.println("Hello");
        Sample s=new Sample();
        s.m1();
    }
}
```

Constructor is not allowed any return type, by mistake if we written any return type to constructor then there is no compile time error and no runtime exception that is treated as normal method.

Exp:-

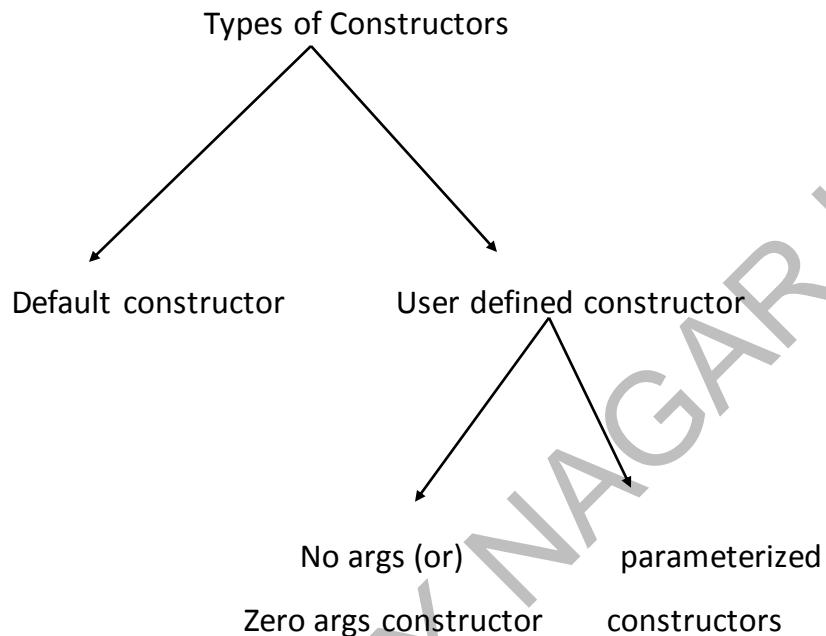
```
class Sample
{
    Void m1 () // It treated as a normal method.
    {
    }
}
```

Note: Constructor is executed exactly at the time of an object creation time. Due to this reason it is recommended to maintain the data base connection establishment code and file connection establishment code inside a constructor.

Types of Constructors:

Java tech has supported two types of constructors.

- 1) Default constructor
- 2) User defined constructor



User defined constructor:

User defined constructor are constructed by programmer based on the application requirement.

Zero (or) No args constructor:

If we are not providing even a single parameter to the user defined constructor we can called as Zero (or) No args constructor.

Exp:-

```
Class sample
{
    Sample()
}
```

```
}
```

Parameterized constructor:

If we are providing at least one parameter to the user defined constructor then we can call it as parameterized constructor.

Exp:-

```
Class sample
{
    Sample(int a)
}
```

Default constructor:

It is provided by the compiler when we are not providing a single constructor to the respective class. Then compiler will append the default constructor.

Exp:-

```
Class sample
{
}
// Here compiler will append the default constructor like (Sample)
```

Note: Every java class contains at least one constructor either programmer provided (or) compiler constructed.

Exp:-

```
class Sample
{
    Sample(int a)
```

```
{  
    System.out.println(a);  
    System.out.println("sample class constructor ");  
}  
  
public static void main(String[] args)  
{  
    Sample s=new Sample();  
    Sample s=new Sample(10);  
}
```

Inheritance

Getting the properties and behaviors from one class to another class is called as inheritance. the inheritance is also called as "Is-a" relationship.

To achieve the inheritance we should use the extends keyword. (Extends means getting) The main advantage of inheritance to produce the code reusability.

Types of inheritance's:

Single inheritance

Multiple inheritance (java is not supported).

By combining the single and multiple inheritances we can derive the different inheritances concepts.

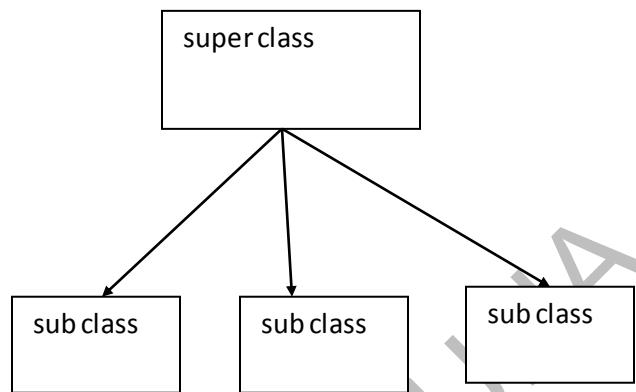
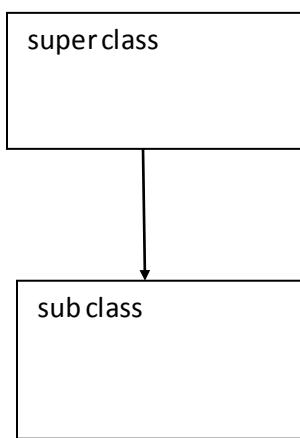
Multilevel inheritance

Hybrid inheritance.(java is not supported).

Single inheritance:

Getting the properties and behaviors from one super class to one or more subclasses are called as single inheritance.

super class is called as base class (or) parent class subclass is called as derived class (or) child class



Exp:-

```
class Example
{
    void m1()
    {
        System.out.println("Example m1");
    }
    void m2()
    {
        System.out.println("Example m2");
        m1();
    }
}

class Sample extends Example
{
    void m1()
    {
        System.out.println("Sample m1");
    }
}

class SpecialCase

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

```
        Sample s = new Sample();

        s.m1(); //Sample m1

        System.out.println();

        s.m2(); // Example m2
                // Example m1
    }
}


```

Exp:-

```
class Parent
{
    public void m1()
    {
        System.out.println("m1-method");
    }
}

class Child extends Parent
{
    public void m2()
    {
        System.out.println("m2-method");
    }
}

class Demo
{
    public static void main(String[] args)
    {
        Parent p=new Parent();
        p.m1(); //valid
        p.m2(); //invalid

        Child c=new Child();
        c.m2(); //valid
        c.m1(); //valid

        Parent p=new Child();
        p.m1(); //valid
        p.m2(); //invalid C.T.E
    }
}
```

```
        Child c=new Parent(); //incomptable type  
    }
```

Note: In java it is possible to hold child class object with the parent reference variable. but by using that reference variable we are not allowed to call child class specific methods.

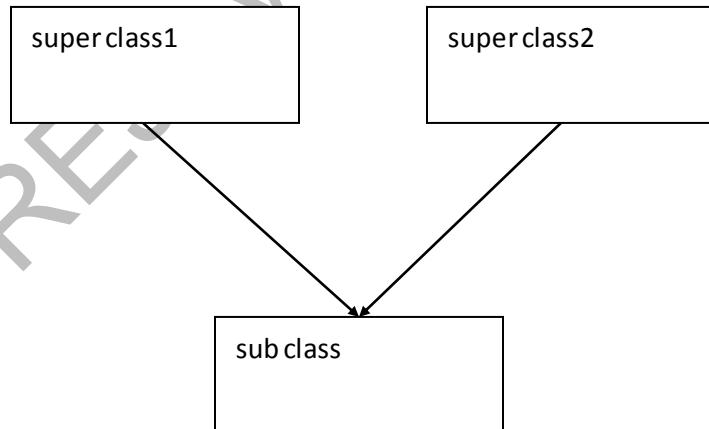
Always the inheritance is achieved parent to child but not child to parent. Java does not support cyclic inheritance.

Exp:-

```
class Parent extends Child  
{  
}  
class Child extends Parent  
{  
}  
// invalid
```

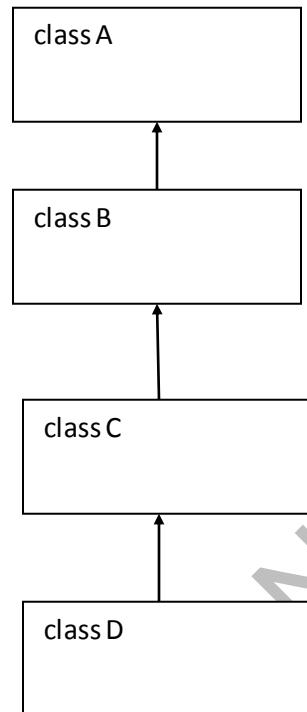
Multiple inheritance:(java is not supported).

Getting the properties and behaviors from more than one super class to one or more subclasses is called as multiple inheritance.



Multilevel inheritance:

combination of single inheritance more than one level is called as multilevel inheritance.



Exp:-

```
class A
{
    public void m1()
    {
        System.out.println(" A-class zero constructor");
    }
}

class B extends A
{
    public void m2()
    {
        System.out.println("B-class zero constructor");
    }
}

class C extends B
{}
```

```

        public void m3()
        {
            System.out.println("C-class zero constructor");
        }
    }

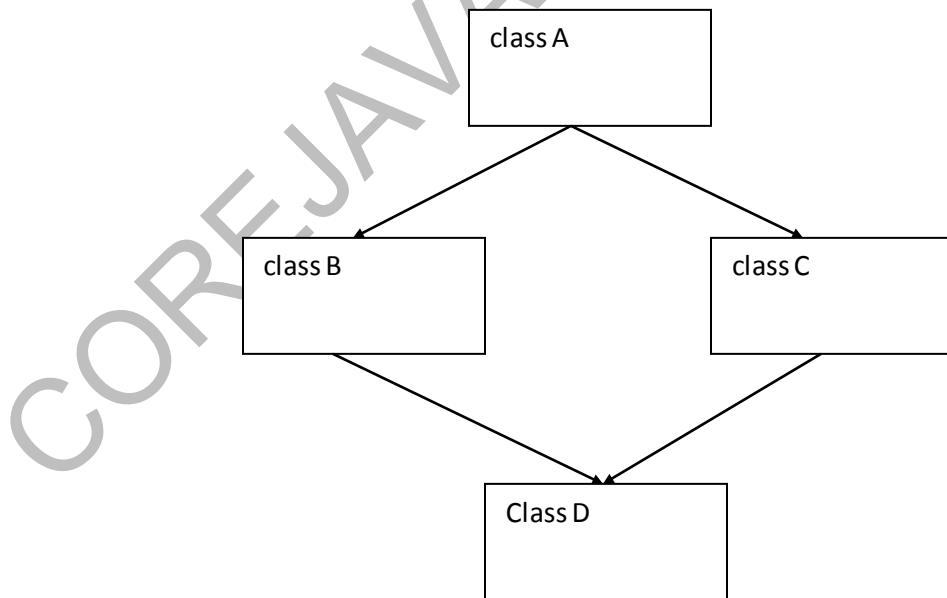
    public static void main(String [] args)
    {

        A a=new A();
        a.m1();
        B b=new B();
        b.m1();
        b.m2();
        C c=new C();
        c.m1();
        c.m2();
        c.m3();
    }
}

```

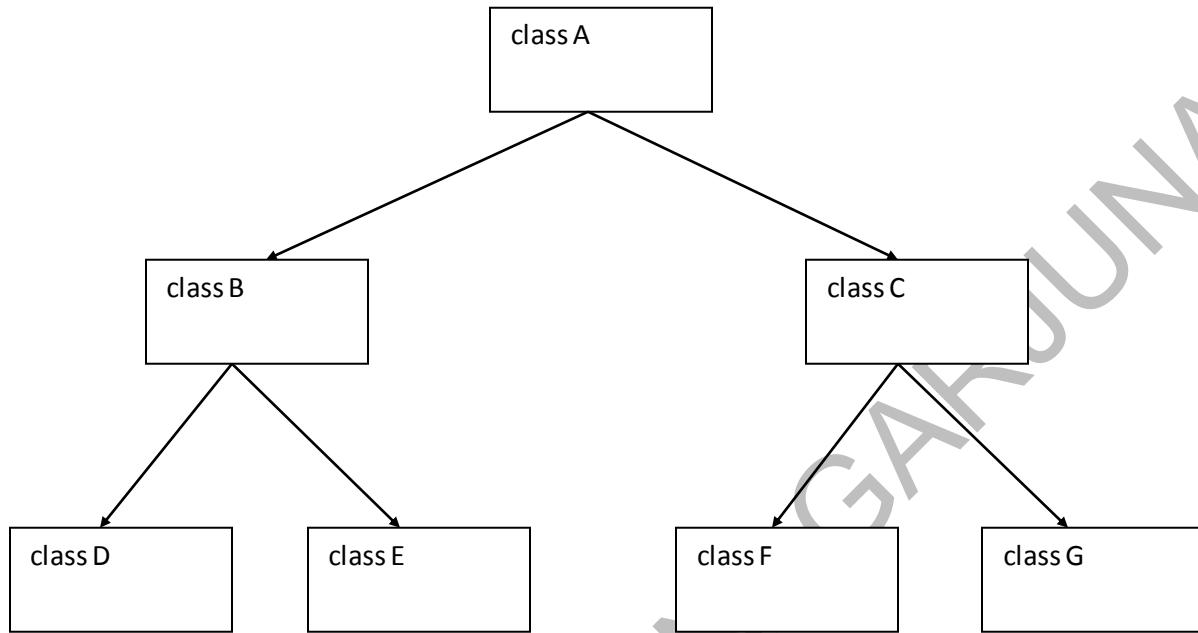
Hybrid inheritance: (Java is not supported)

The combination of single and multiple inheritance is called as hybrid inheritance.



Hierachal Inheritance:

The combination of the single inheritance in a tree structured manner is called as hierachal inheritance.



Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Exp:-

```
class A{  
    void msg()  
    {  
        System.out.println("Hello");  
    }  
}  
  
class B  
{  
    void msg()  
    {  
        System.out.println("Welcome");  
    }  
}
```

```
}

}

class C extends A,B{//suppose if it were

public static void main(String args[])

{

C obj=new C();

obj.msg();//Now which msg() method would be invoked?

}

}
```

Abstract

Abstract is modifier it is applicable for classes and methods. it is not applicable for variables. Both abstract class and abstract method should be declared by using the key word "abstract".

Abstract Method:

An abstract method is a method without method body. An abstract method is written when the same method has to perform the different task depending upon the object calling it.

- a. abstract method does not contain any method body
- b. abstract method always ends with semicolon only.

Exp:-

```
abstract class AbstExp
{
abstract void calculate(double x);
}

class Sub1 extends AbstExp
{
void calculate(double x)
{
    System.out.println("square="+ (x*x));
}
}
```

```
class Sub2 extends AbstExp
{
void calculate(double x)
{
    System.out.println("cube="+ (x*x*x));
}
}

class Demo
{
    public static void main(String[] args)
    {
        Sub1 s=new Sub1();
        s.calculate(10);
        Sub2 s1=new Sub2();
        s1.calculate(10);
        AbstExp ref;
        ref=s;
        ref.calculate(10);
        ref=s1;
        ref.calculate(10);
    }
}
```

Exp:-

```
abstract class A
{
public abstract void m1()
{
    System.out.println("Hi");
}
}
```

C.T.E: abstract methods cannot have a body
public abstract void m1()

Exp:-

```
abstract class A
{
    public abstract void m1();
}
```

```
    }
}
// valid
```

Abstract class:

- An abstract class is allowed zero (or) more no of abstract methods and zero (or) more no of concrete methods. (complete methods)

Exp:-

```
abstract class Test
{
}
}//valid
```

Exp:-

```
abstract class Test
{
    abstract void m1();
    public void m2()
    {
        System.out.println("Hello");
    }
}
```

Exp:-

```
abstract class Test
{
    void m1()
    {
    }
    void m2()
    {
    }
    void m3()
    {
    }
}      //valid it contains only concrete methods.
```

- Object creation is not possible to abstract class why because abstract class contains incomplete methods , it is not possible to estimate the total memory required to create the object. So, JVM can't create object to an abstract class.

We should create the sub classes and all the abstract methods should be implemented (body should be written) in the sub classes . Then it is possible to create the object to the sub classes, But variable creation is possible.

Exp:-

```
abstract class AA
{
    abstract void m1();
}

class B extends AA
{
    public void m1()
    {
        System.out.println("Hi");
    }

    public static void main(String[] args)
    {
        System.out.println("Hello");
        AA a=new AA();
// C.T.E: AA is abstract; cannot be instantiated
        AA a=new AA();
        ^
        a.m1();
        B b=new B();
        b.m1();
    }
}
```

- An abstract class is allowed constructors, non-static variables, non-static methods, non-static blocks, static variables, static methods, static blocks.

Note: An abstract class is allowed constructor to provide initialization to the non-static variables.

If any class contains at least one abstract method that class must be declare with abstract otherwise we will get compile time error.

Exp:-

```
class Test
{
    abstract void m1();
```

```
}
```

C.T.E: Test.java:1: Test is not abstract and does not override abstract method m1() in Test

```
class Test
```

```
^
```

Exp:-

```
abstract class Test
{
    public abstract void m1();
}
// valid
```

Abstract modifier and following modifiers combination is illegal for methods.

private abstract void m1();

private methods are not visible in child class, overriding is not possible to private methods.

abstract methods means must be override in the child class and provide the implementation.

final and abstract:

```
final abstract void m1();
```

final means overriding is not possible, abstract methods means must be override in the child class and provide the implementation.

Strictfp and abstract:

```
strictfp abstract void m1();
```

strictfp is always talks about method implementation. But abstract always talks method implementation in subclass.

strictfp and abstract combination legal for classes but illegal for methods.

Exp:-

```
strictfp abstract class A
{
    //valid
}
```

native and abstract:

Syn:- native abstract void m1();

native modifier is applicable for only for methods, but not for variables and classes & interfaces.

native methods are developed by some other languages like C, C++.

In java always the native methods are ends with the semicolon(;) because native methods already having method body.

Exp:-

```
class Test
{
    native void m1(); //invalid
    native void m2() //valid
}
```

native methods is always decrease the platform independency, where as strictfp is always increase the platform independency.(it follows IEEE rules given then it gives exact floating point reset).

synchronized and abstract:

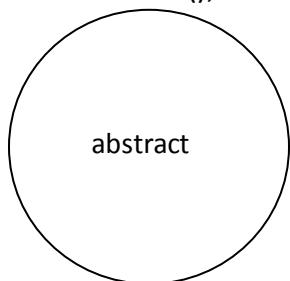
Syn:- synchronized abstract void m1();

if any modifier talks about method implementation, that modifier and abstract modifier combination is always illegal for methods.

synchronized modifier allowed methods and blocks not applicable for variables and classes and interfaces.

static and abstract:

static abstract void m1();



private
final
static
synchronized

strictfp

native

above specie abstract modifier and all modifiers combination is always illegal

NOTE: transient and volatile modifiers are applicable for only variables but not for methods and classes.

Interface

We can define an interface in different perspectives from the client point of view an interface defines the set of services what he is expecting.

From the service provider point of view an interface defines the set of services what he offering hence interface act as a contract b/w client and service provider.

100% pure abstract class is also called as an interface.

Any requirement specification is also defined as an interface.

Java Interface:

An interface is allowed zero (or) more number of abstract methods only.

Exp:- an interface is contains only method declaration.

```
interface Test
{
    public void m1(); // valid
}
```

Exp:-

```
interface Test
{
    public void m1()
    {
        System.out.println("Hi");
    }
}
```

```
D:\>javac Test9.java
```

```
Test9.java:4: error: interface methods cannot have body
```

```
{}
```

Why we use Interface:

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

- By using interface keyword we can define the interface.
- By using implement keyword we can implement an interface provide the services. (Implementation to methods).

Exp:-

```
interface interface name  
{  
    //variables  
    //methods  
}
```

Exp:-

```
interface Test  
{  
    int a=10;  
    int b=30;  
    void add();  
    void sub();  
}
```

```
D:\>javac Test.java

D:\>javap Test

Compiled from "Test.java"

interface Test

{

    public static final int a;

    public static final int b;

    public abstract void add();

    public abstract void sub();

}
```

By default all the interface variables are public static final.

By default all the interface methods are public and abstract.

Exp:-

```
interface Test

{

    public static final a=20;

    public static final b=20;

    public abstract void add();

    public abstract void sub();

}
```

An interface is not allowed constructors, static-blocks, static-methods, and instance blocks and instance variables.

If we are declaring any non-static variables inside a interface that is also be converted into static variables.

Instantiation is not possible to interface but reference variable creation is possible.

Exp:-

```
interface ShapeEx
{
    void m1();
}

class Sample5 implements ShapeEx
{
    public void m1()
    {
        System.out.println("Hi");
    }

    public static void main(String[] args)
    {
        ShapeEx e=new ShapeEx();
        // D:\>javac Sample5.java
        //Sample5.java:13: error: ShapeEx is abstract; cannot be instantiated
        Sample5 s=new Sample5();
        s.m1();

        ShapeEx e1= new Sample5();
        e1.m1();
    }
}
```

Difference B/W abstract class and interface:

Abstract class

- 1) An abstract class is allowed zero (or) more abstract methods and zero

Interface

- 1) An interface is allowed zero (or) more no of abstract methods only.

- (or) more concrete methods.
- 2) Inside an abstract class there is no Default values for methods and Variables.
 - 3) Inside an abstract class is allowed Constructor, static blocks, methods, Variables, non- static blocks, methods, Variables.
 - 4) Instantiation is not possible for Abstract class.
 - 5) Reference variable creation is Possible for abstract class.
 - 6) Abstract class is declared by Using the keyword abstract.
- 2) Inside an interface all are default cases by default all the variables are public Static final.
 - 3) Inside an interface its not allowed to static blocks, methods, non static blocks
 - 4) Instantiation is not possible for an Interface.
 - 5) Reference variable creation is possible For an interface.
 - 6) Interface is declared by using the keyword interface

An interface is not allowed private methods and protected methods and private, protected variables.

Exp:-

```
interface Test
{
    private int x=38; // invalid
}
```

C.T.E: D:\>javac Test38.java

error: modifier private not allowed here

Exp:-

```
interface Test
{
    private void m1(); // invalid
    protected void m2(); // invalid
```

```
}
```

D:\>javac Test9.java

Test9.java:3: error: modifier private not allowed here

```
private void m1(); // invalid
```

Test9.java:4: error: modifier protected not allowed here

```
protected void m2(); // invalid
```

- An interface is allowed only four modifiers.

Public

<default>

Abstract

Strictfp

Exp:-

```
public interface Test
{
    // valid
}
```

Exp:-

```
final interface Test
{
    // invalid
}
```

Exp:-

```
abstract interface Test
{
    // valid
}
```

```
}
```

Exp:-

```
Strictfp interface Test
{
    // valid
}
```

Note:

When ever a class implements an interface compulsory we should provide implementation for every method in that class, otherwise we can declare that implementation class as a abstract otherwise we will get the compile time error.

Exp:-

```
interface Test9
{
    void m1();
    void m2();
}

class Service implements Test9
{
    public void m1()
    {
        System.out.println(" Hello ");
    }
}
```

```
D:\>javac Test9.java
```

```
Test9.java:6: error: Service is not abstract and does not override abstract
method m2() in Test9
```

```
class Service implements Test9
```

Exp:-

```
interface Test9
```

```
{  
void m1();  
void m2();  
}  
  
abstract class Service implements Test6  
{  
public void m1()  
{  
System.out.println(" Hello ");  
}  
}
```

Exp:-

```
interface Test9  
{  
void m1();  
void m2();  
}  
  
abstract class Service implements Test9  
{  
public void m1()  
{  
System.out.println(" Hello ");  
}  
}  
  
class Service1 implements Test9  
{  
public void m1()  
{
```

```
        System.out.println(" hi ");
    }
    public void m2()
    {
        System.out.println(" h r u ");
    }
    public static void main (String[] args)
    {
        Service1 s=new Service1();
        s.m2();
        s.m1();
    }
}
```

Whenever we are implementing class we must be maintain the interface methods has a public otherwise we will get the compile time error. (Because weak access privileges).

Exp:-

```
interface Test9
{
    void m1();
    void m2();
}
abstract class Service implements Test9
{
    public void m1()
    {
        System.out.println(" Hello ");
    }
}
class Service1 implements Test9
```

```
{  
    void m1()  
    {  
        System.out.println(" hi ");  
    }  
    public void m2()  
    {  
        System.out.println(" h r u ");  
    }  
    public static void main (String[] args)  
    {  
        Service1 s=new Service1();  
        s.m2();  
        s.m1();  
    }  
}
```

Interface variables:

By default the interface variables are **public static final**

Exp:- interface Test

```
{  
    public static final int x=39;  
}
```

Why we declare variable as public:

We can access the interface variables from anywhere in the implementation classes.

Why we declare variable as static:

Access the interface variables without creating the object in the implementation classes.

Why we declare variable as final:

We can use the interface variable in the implementation classes but we can't modify the interface variables in the implementation classes, why because interface is a client requirement specification.

Note: Interface variables must be initialized at the time of creation, otherwise we will get the compile time error.

Exp:-

```
interface Test
{
    int x; //invalid
}
```

Exp:-

```
interface Test9
{
    int x;
    void m1();
}

class Service3 implements Test9
{
    public void m1()
    {
        System.out.println(" Hi ");
    }
    public static void main(String[] args)
    {
        Service3 s=new Service3();
        s.m1();
        System.out.println(s.x);
    }
}
```

```
}
```

D:\>javac Service3.java

Service3.java:3: error: = expected

```
    int x;
```

Exp:-

```
interface Test
{
    int x=40; //valid
}
```

If two interfaces contains variable with same name, if we are implemented two interfaces simultaneously then there may a chance of getting the variable naming conflicts but we can resolve by interface variable naming conflict by using the respective interface names.

Exp:-

```
interface Test9
{
    int x=40;
}
interface Test10
{
    int x=39;
}
class Service3 implements Test9, Test10
{
    public static void main(String[] args)
    {
        System.out.println(x);
    }
}
```

/D:\>javac Service3.java

```

//Service3.java:14: error: reference to x is ambiguous, both variable x in Test9 and
variable x in Test10 match System.out.println(x);

    Service3 s=new Service3();

    System.out.println(s.x);

//D:\>javac Service3.java Service3.java:18: error: reference to x is ambiguous, both
variable x in Test9 and variable x in Test10 match System.out.println(s.x);

    System.out.println(Test9.x);

    System.out.println(Test10.x);

D:\>javac Service3.java

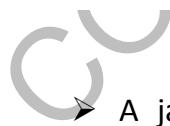
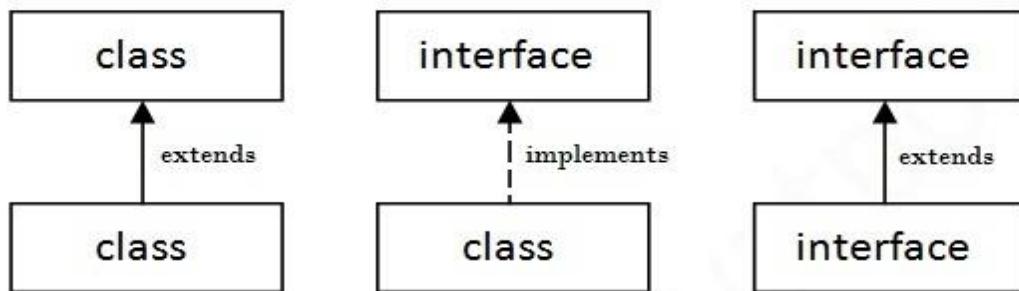
D:\>java Service3

40 39

}

```

Extends and implements:



- A java class can extend only one class at a time, java class need not extend multiple classes simultaneously.
- A java class can implement an interface.
- A java class can extend a class and Implements an interface.

Exp:-

```
class A
{
interface X
{
class B extends A implements X
{
}
```

- Java class implements multiple interfaces simultaneously.
- Java class can extend a class and implements multiple interfaces simultaneously.

Exp:-

```
class A
{
interface X
{
}
interface Y
{
}
class B implements X,Y
{}
```

Java interface extend another interface

Exp:-

```
interface A
{
}
interface B extends A
```

```
{  
}
```

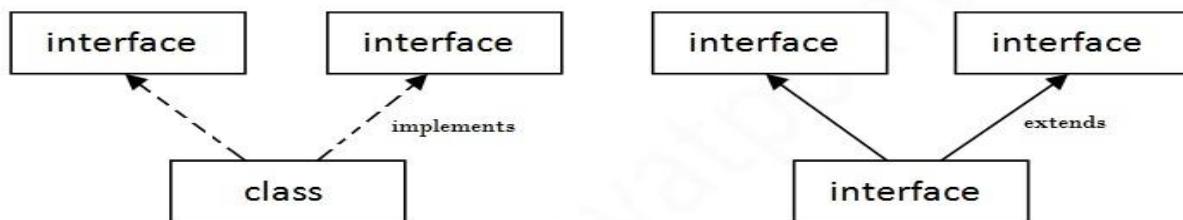
Java interface extends multiple interfaces.

Exp:-

```
interface A  
{  
}  
}  
interface B  
{  
}  
}
```

```
interface C extends A,B  
{  
}  
}
```

Java supports multiple inheritances through the interface. If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

Exp:-

```
interface Printable{  
    void print();  
}  
interface Showable{  
    void show();  
}
```

```
}

class Mul implements Printable,Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}
    public static void main(String args[]){
        Mul m= new Mul();
        m.print();
        m.show();
    }
}
```

Output: Hello

Welcome

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Exp:-

```
class A
{
    void msg()
    {
        System.out.println("Hello");
    }
}
```

```
class B
{
    void msg()
    {
```

```
        System.out.println("Welcome");
    }
}

class C extends A,B

{//suppose if it were

public static void main(String[] args);

{
    C obj=new C();

    obj.msg();//Now which msg() method would be invoked?
}
}
```

Q) Multiple inheritance is not supported in case of class but it is supported in case of interface, why?

As we have explained in the inheritance chapter, multiple inheritances is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

Exp:-

```
interface Printable
{
    void print();
}

interface Showable
{
    void print();
}

class Mul2 implements Printable,Showable
{
```

```
public void print(){System.out.println("Hello");  
}  
public static void main(String args[]){  
    Mul m=new Mul();  
    Mul.print();  
}
```

- An interface can't implement another interface.

Exp:-

```
interface A  
{  
}  
  
interface B implements A  
{  
    //invalid  
}
```

Interface method naming conflicts:

case1:

If the two interfaces contains same method signature and return type, if we are implemented that interfaces simultensioly just we can provide the implementation for only one method.

Exp:-

```
interface Left  
{  
    public void m1();  
}  
  
interface Right
```

```
{  
    public void m1();  
}  
  
class TestVNC implements Left,Right  
{  
    public void m1()  
    {  
        System.out.println("Hi");  
    }  
    public static void main(String[] args)  
    {  
        TestVNC t=new TestVNC();  
        t.m1();  
    }  
}
```

case2:

If the two interfaces contains the same method signature and the different return type. If it is not possible to implement the two interfaces simultaneously .

Exp:-

```
interface Left  
{  
    public void m1();  
}  
  
interface Right  
{  
    public int m1();  
}  
  
class TestVNC implements Left,Right
```

```
{  
    public void m1()  
    {  
        System.out.println("Hi");  
    }  
    public static void main(String[] args)  
    {  
        TestVNC t=new TestVNC();  
        t.m1();  
    }  
}
```

OutPut:

```
D:\>javac TestVNC.java  
TestVNC.java:10: error: TestVNC is not abstract and does not override abstract m  
ethod m1() in Right  
class TestVNC implements Left,Right  
^  
TestVNC.java:13: error: m1() in TestVNC cannot implement m1() in Right  
    public void m1()  
    ^  
    return type void is not compatible with int  
2 errors
```

Q) It is possible to implement multiple interface simultaneously ?

Yes, except one case that is if the two interfaces contains same method signature and different return types.

What is marker or tagged interface: An interface that has no member is known as marker or tagged interface. For example: **Java.lang.Serializable, java.io.Cloneable**, etc.

They are used to provide some essential information to the JVM so that JVM may perform some useful operation. It is also called as tag interface.

- By implementing Serializable interface our object can travel across the network.
- By implementing Cloneable interface our object can provide exactly duplicated object.

Exp:-

```
public interface Serializable
{
}
```

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many information such as id, name, emailed etc. It contains one more object named address, which contains its own information such as city, state, country, zip code etc. as given below.

Exp:-

```
class Employee{
    int id;
    String name;
    Address address;//Address is a class
}
```

Why use Aggregation:

For Code Reusability.

When use Aggregation:

Code reuse is also best achieved by aggregation when there is no is-a relationship. Inheritance should be used only if the relationship is-a is maintained

throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

In this example, Employee has an object of Address, address object contains its own information's such as city, state, country etc. In such case relationship is Employee HAS-A address.

Exp:- **Address.java**

```
public class Address {  
    String city,state,country;  
    public Address(String city, String state, String country) {  
        this.city = city;  
        this.state = state;  
        this.country = country;  
    }  
}
```

Emp.java

```
public class Emp {  
    int id;  
    String name;  
    Address address;  
    public Emp(int id, String name,Address address) {  
        this.id = id;  
        this.name = name;  
        this.address=address;  
    }  
    void display(){
```

```
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}
public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
e.display();
e2.display();
}
}
```

Output: 111 varun

gzb UP india

112 arun

gno UP india

Access Modifiers

Access modifiers are used to define the accessibility scope to our java program elements.

Java tech has supported four access modifiers.

- 1) Public
- 2) Private
- 3) Protected
- 4) <default>

Private:

Private modifier is not applicable for top-level classes (outer classes). It is applicable for variables, methods, constructor and inner classes. The private access modifier is accessible only within class.

If we are declaring any member as a private we can access that member with in the same class only .so private modifier is also called class level modifier.

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

Exp:-

```
class A
{
    private int data=40;
    private void msg()
    {
        System.out.println("Hello java");
    }
}
public class Simple
{
    public static void main(String args[])
    {
        A obj=new A();
        System.out.println(obj.data);//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

Data hiding:

Out side person is not allowed to access the data directly. We can achieve data hiding by using the private modifier the main advantage of data hiding is to provide security.

Exp:-

```
class Test
{
    private int data=40;
    private void msg()
}
```

Public:

Public modifier is applicable for classes, interfaces, constructors, methods, variables (local variable is not allowed) and inner classes.

If we are declaring any class as a public we can access the class from anywhere that is within the same package and outside of the package.

If we are declaring any member as a public we can access that member from anywhere. But before checking the member visibility we check class visibility.

Exp:-

```
//save by A.java

package pack;
public class Test
{
    public void msg()
    {
        System.out.println("Hello");
    }
}

//save by B.java
```

```
package mypack;
import pack.*;
```

```
class Demo
{
    public static void main(String args[])
    {
        Test obj = new Test();
        obj.msg();
    }
}
```

Protected:

If we are declaring any member as a protected we can access that member with in the current package anywhere and from outside package only child class (child class reference).

We can access protected members with in the current package any where either by using child reference (or) by using parent reference.

By outside package we can access protected members only child classes, and we must be use child reference only.

Exp:-

```
//save by A.java
package pack;
public class A
{
    protected void msg(){System.out.println("Hello");
}
}

//save by B.java
package mypack;
```

```
import pack.*;  
  
class B extends A  
{  
  
    public static void main(String args[])  
    {  
  
        B obj = new B();  
  
        obj.msg();  
  
    }  
  
}
```

<default>:

If we are not using any access modifier to a class and members are called <default> members.

<default> access modifier is not a keyword, we can access the <default> class and members with in the current package only.

<default> access modifier is called package level access modifier

Exp:-

```
//save by A.java  
  
package pack;  
  
class A  
{  
  
    void msg()  
    {  
  
        System.out.println("Hello");  
  
    }  
  
}
```

```
}

//save by B.java

package mypack;

import pack.*;

class B

{

    public static void main(String args[])

    {

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

        obj.msg();//Compile Time Error

    }

}
```

Polymorphism

polymorphism is a Greek word 'poly' means many and 'morphism' means forms.

The ability to exist in different forms is called polymorphism in java. A variable and method can exist in different forms.

The main advantage of polymorphism is to provide the flexibility.

They are two types of polymorphisms in java

- 1) compile time polymorphism (or) static polymorphism.
- 2) runtime polymorphism (or) dynamic polymorphism.

Compile time polymorphism:

The polymorphism exhibited at compilation time is called static polymorphism (or) compile time polymorphism.

Exp:-

Method overloading

Method hiding

In static polymorphism java compiler is identified which method is called, of course JVM executed the method later.

Here java compiler know and bind the method call with method code (body) at the time of compilation. so it is called as static binding (or) compile time polymorphism.

Run time polymorphism:

The polymorphism exhibited at run time is called dynamic polymorphism (or) run time polymorphism.

Exp:-

Method overriding

In dynamic polymorphism java compiler does not know which method is called at the time of compilation. only JVM decides which method is called at run time.

Method overloading:

We can define more than one method with the same name and different parameters in a class is called as method overloading.

In method overloading the method resolution (identification) is always take care by compiler based on reference type (or) argument list. It is called as a compile time polymorphism (or) static polymorphism (or) early binding.

Exp:-

```
public class Poly  
{  
    public void m1()  
    {  
        System.out.println("zero arg m1");  
    }  
}
```

```
}

public void m1(int i)

{

System.out.println("int arg m1");

}

public void m1(float f)

{

System.out.println("float arg m1");

}

public static void main(String[] args)

{

Poly p=new Poly();

p.m1();

p.m1(10);

p.m1(10.5f);

p.m1(10);

}

}

/*output

zero arg m1

int arg m1

float arg m1

int arg m1*/
```

Automatic promotion in method overloading:

In method overloading the method resolution (identification) is take care by compiler. If the compiler unable to find a method with the required argument compiler won't raise any compile time error immediately. First compiler promotes to next level and checks for required method if it is not available then once again promotes to next level to check for required method this process will be continue until all possible promotions, still if the compiler unable to find the required methods then compiler will raise the compile time error.

Case 1: Two arguments at same level

In method overloading if the two arguments are at same level always we will get the compile time error.

Exp:-

```
public class Poly1
{
    public void m1(int i,float f)
    {
        System.out.println("int arg,float arg ");
    }
    public void m1(float f,int i)
    {
        System.out.println("float args,int args");
    }
    public static void main(String[] args)
    {
        Poly1 p=new Poly1();
        p.m1(20,20);
    }
}
```

```
}

}

/*output

Error: Poly1.java:14: reference to m1 is ambiguous, both method m1(int,float)
in Poly1

and method m1(float,int) in Poly1 match

p.m1(20,20);

*/
```

Case2:

If the arguments having the " IS-A" relationship then always child class will get the height priority.

Exp:-

```
class Animal

{

}

class Monkey extends Animal

{

}

public class IsaRelation

{

    public void m1(Animal a)

    {

        System.out.println("animal version");

    }
```

```
public void m1(Monkey m)
{
    System.out.println("monkey version");
}

public static void main(String[] args)
{
    IsaRelation a=new IsaRelation();
    //a.m1(new Animal());
    //a.m1(new Monkey());
    //a.m1(null);// monkey child will get height priority
    Animal a1=new Monkey();
    a.m1(a1);// Animal (it will depend up on the reference variable "a1" parameter in
    overloading)
}
```

In method overloading the method resolution is always take care by compiler based on reference and argument list.

Method Overriding:

Method Signature:

Method signature means method name and followed by parameter list.

In method signature order of the parameters are important. return type is not part of method signature.

```
public void m1(int i, float f)
```

Overriding:

What ever the parent having by default available to the child.

If child was not satisfied with parent implementation then child is allowed to override the parent class implementation in its own way this concept is nothing but overriding.

Method Overriding:

In method overriding method names and parameters must be same including in both the parent and child.

Exp:-

```
class Parent
{
    public void property()
    {
        System.out.println("land,gold,money");
    }

    public void marry()
    {
        System.out.println("marry");
    }
}

class Child extends Parent
{
    public void marry()
    {
        System.out.println("xxx");
    }
}
```

```
public class MethdOverriding  
{  
    public static void main(String[]args)  
    {  
        Parent p=new Parent();  
        p.property();  
        p.marry();  
  
        Child c=new Child();  
        c.property();  
        c.marry();  
  
        Parent p1=new Child();  
        p1.property();  
        p1.marry();  
    }  
}
```

Output:

land,gold,money

marry

land,gold,money

xxx

land,gold,money

xxx

In method overriding method resolution is always take care by JVM based on the runtime object. Hence method overriding is consider as a dynamic polymorphism (or) run time polymorphism (or) late binding.

Rules for method overriding:

Rule1:

private methods are not visible in the child class. hence overriding concept is not applicable for private methods, based on application requirement we can take exactly what method there in super class we can write same method in child class then it is valid but it is not a method overriding concept.

Exp:-

```
class A
{
    private void m1()
    {
    }
}
class B extends A
{
    private void m1()
    {
    }
}
```

Rule2:

while overriding weaker access modifiers are not allowed.

parent -----> public

child -----> public // valid

public -----> protected/<default>/private //invalid

protected -----> protected/public //valid

protected -----> <default>/private //invalid

<default> -----> <default>/protected/public //valid

<default> -----> private // invalid

Exp:-

```
class Parent
{
void m1()
{
}
class Child extends Parent
{
private void m1()
{
}
}
```

// invalid

parent -----> private

child -----> <default>/protected/public/private

// valid but not overriding why because parent property (or) method is not visible to child class.

Rule3:

While overriding return type is must be same but this rule is applicable only up to java 1.4 version. But from java 5.0 version on words covariant return type also allowed.

According to child method return type need not be same as parent method return type, but its child classes also allowed.

parent return type :

child return type:

- 1) Object -----> String //valid
- 2) Number -----> Integer //valid
- 3) String -----> Object //invalid
- 4) double -----> int //invalid (**primitives are not part of the covariants**)

Rule4:-

final is modifier is applicable for variables, methods and classes.

- a) final variable means reassignment is not possible.

Exp:-

```
final int x=39;  
int x=30; //C.T.E
```

can't assign a value to final variable x

```
class Test  
{  
final int x;  
public static void main (String[] args)  
{  
Test t=new Test();  
t.x=39; // C.T.E  
}  
}
```

- b) final method means overriding is not possible.

Exp:-

```
class Test  
{  
public final void marry()  
{  
System.out.println("Hi");  
}  
class Test1 extends Test  
{  
public void marry()  
{  
System.out.println("Hello");  
}  
}
```

```
// marry in Test1 can't override marry in the Test : overridden method is final
```

c) final class means inheritance is not possible.

Exp:

```
final class Parent  
{  
}  
  
class Child extends Parent  
{  
}
```

Method hiding:

Method hiding is similar to method overriding concept except the following difference.

In method hiding both the methods (parent , child) should be static.

In method overriding both the methods should be non-static.

In method hiding method resolution always take care by compiler based on object reference type and parameter list. Hence method hiding is consider as static polymorphism (or) compile time polymorphism.

In method overriding method resolution is always take care by JVM based on run time object. Hence method overriding consider as dynamic polymorphism (or) run time polymorphism.

Exp:-

```
class Parent
```

```
{
```

```
public static void marry()
```

```
{
```

```
        System.out.println("marry");

    }

}

class Child extends Parent

{

public static void marry()

{

    System.out.println("xxx");

}

}

public class MethdHiding

{

public static void main(String[]args)

{

    Parent p=new Parent();

    p.marry();

    Child c=new Child();

    c.marry();

    Parent p1=new Child();

    p1.marry();

}

}
```

Output:

marry

xxx

marry

Var argument method:

Until java4.0 version we can't declare a method which can taken variable number of arguments.

If there is any change in no of arguments compulsory we should go for new method. It increase the length of the code and the reduce the readability.

We can resolve this problem by using "var argument" methods from java5.0 version on words.

We can declare a method which take variable no of arguments such type of methods are called "var argument" methods.

Syntax:

Method name (data type... var name)

We can invoke this method (var args) by passing any no of arguments including zero no of arguments also. Always "**var args**" method gets **least priority**.

Exp:-

```
class SampleVar
{
    int i;
    public void m1(int i)
    {
        System.out.println("int args m1"+i);
    }
    public void m1(int...x)
    {
        System.out.println("var args m1");
        for (i=0;i<x.length ;i++ )
    }
```

```
        System.out.println(x[i]);  
    }  
}  
  
public static void main(String[] args)  
{  
    SampleVar s=new SampleVar();  
    s.m1();  
    s.m1(10);  
    s.m1(10,30);  
    s.m1(10,20,30);  
    s.m1(10,20,30,40,50);  
}
```

Output:

```
D:\>javac SampleVar.java  
D:\>java SampleVar  
var args m1(zero args )  
int args m1 10  
var args m1  
10  
30  
var args m1  
10  
20  
30  
var args m1  
10  
20
```

30

40

50

Encapsulation

Encapsulation is the process of wrapping up the data members and associated methods into a single unit is called as encapsulation.

The process of binding the data members and methods into a single unit is called an encapsulation. The main advantage of encapsulation providing the security.

- Every java class follows the encapsulation mechanism.
- Every java package follows the encapsulation mechanism.

Note:

If we take any class we write the variables and methods inside a class. That class is binding the variables and methods so class is example for encapsulation.

In the project generally the variables of class is declared by using private modifier to provide the security.

Generally the methods of a class is declared by using the public, this means the public methods we can use any where (with in the package and outside the package).

So to using the variables from outside we should take help of the public methods.

There is no another way to interacting the private variables.

Advantage of Encapsulation

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

Simple example of encapsulation in java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
//save as Student.java  
  
package com.javatpoint;  
  
public class Student{  
  
    private String name;  
  
  
    public String getName(){  
  
        return name;  
    }  
  
    public void setName(String name){  
  
        this.name=name  
    }  
}  
  
//save as Test.java  
  
package com.javatpoint;  
  
class Test{  
  
    public static void main(String[] args){  
  
        Student s=new Student();  
  
        s.setName("vijay");  
  
        System.out.println(s.getName());  
    }  
}
```

```
}
```

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay

Abstraction

The process of hiding the internal implementation logic and just highlighting the set of services what we are offering is called an abstraction. main advantage of abstraction is security.

Packages

The set of class's interfaces and sub packages are grouped into a single unit is called package. Package is a folder (or) a directory.

A package is a group of similar types of classes, interfaces and sub-packages.

Advantage of Package

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.

Java tech has supported two types of packages

- 1) Predefined package.
- 2) User defined package.

Predefined package:

These packages are available with along java s/w.

Java.lang package:

The regularly used classes and interfaces are grouped into a single unit that is called Java.lang package.

Java.lang package is by default available to every java program hence is not required to write import statement to Java.lang package.

Java.lang package contains the following classes.

- ❖ All Wrapper classes
- ❖ String, String Buffer, String Builder
- ❖ Thread class
- ❖ Object class
- ❖ System class
- ❖ Exception class
- ❖ Error class
- ❖ Throwable class
- ❖ Class

Interfaces:

- ❖ Runnable
- ❖ Cloneable
- ❖ Comparable

Java.io package:

Java.io package provides classes and interfaces to perform input and output operations.

Classes:

- ❖ Output stream
- ❖ Input stream
- ❖ Buffered input stream
- ❖ Buffered output stream
- ❖ Data input stream
- ❖ Data output stream
- ❖ File writer
- ❖ File reader
- ❖ Buffer writer
- ❖ Buffer reader

Interfaces:

- Serializable (marker interface)
- Data input
- Data output

Java.util package:

Java.util package is used to design the data structure applications.

Classes:

- ArrayList
- LinkedList
- Vector
- Stack
- HashSet
- TreeSet
- HashMap
- IdentityHashMap
- WeakHashMap
- HashTable
- Properties class
- Data class
- Collections.

Interfaces:

- Collection
- Queue (Java5)
- List
- Set
- Iterator
- ListIterator
- Enumeration
- Map
- SortedMap
- NavigableMap (Java6)
- SortedSet
- NavigableSet (Java6)

Java.awt package:

It is used to design GUI applications.

Classes:

- Component
- Frame
- Window
- Panel
- Container
- Color
- Graphics
- Button
- Text field
- Text area
- Label
- List
- Menu bar
- Menu
- Menu item
- Scroll bar

Java.awt.event package:

Event handling classes

- Action event
- Item event
- Adjustment event and so on.

Interfaces:

- Action listener
- Adjustment listener
- Key listener
- Mouse Listener
- Window listener

Javax.swing package:

Classes:

- JFrame

- JButton
- JPanel
- JFilechooser
- JMenu
- JMenuItem
- JComboBox
- JTextField
- JPasswordField

Java. Applet package:

Classes:

Applet class

Java. Text package:

Java. Text package is used to format the numerical values date and time.

Classes:

- Number Format class
- Decimal Format class

Java. Lang. reflect package:

This package is used to develop container s/w and tools. it is also used get the class information.

Classes:

- Method
- Constructor
- Modifier
- Field and

User defined package:

To create user defied package we should use the following

Syntax: package packagename;

Exp:-

```
package pack1;  
  
class Sample  
  
{  
  
    public static void main (String[] args)  
  
    {  
  
        System.out.println("Hello");  
  
    }  
}
```

How to execute the user defined package

D:\>javac -d . Sample2.java (here (.) means current location and - d is destination folder)

D:\>java pack1.Sample2

Hello World!

Data Type Casting (Type Conversion)

Java supports two types of castings – primitive data type casting and reference type casting. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting. Now let us go for data type casting.

Java data type casting supports three types

- ❖ **Implicit casting**
- ❖ **Explicit casting**
- ❖ **Boolean casting.**

1. Implicit casting (widening conversion)(or)Up Casting:

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.

- ❖ the target type is larger than the source type



Exp:-

```
int x = 10;           // occupies 4 bytes
double y = x;         // occupies 8 bytes
System.out.println(y); // prints 10.0
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

```
public class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        long l = i; //no explicit type casting required</b></font>
        float f = l; /no explicit type casting required</b></font>
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

}

2. Explicit casting (narrowing conversion)(or)Down Casting:

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires explicit casting; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

- ❖ When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.



Exp:-

```
public class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        long l = (long)d; //explicit type casting required
        int i = (int)l; //explicit type casting required</b></font>
        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);
    }
}
```

```
double x = 10.5;      // 8 bytes  
  
int y = x;           // 4 bytes ; raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error.
Let us explicitly type cast it.

```
double x = 10.5;  
  
int y = (int) x;
```

The double x is explicitly converted to int y. The thumb rule is, on both sides, the same data type should exist.

3. Boolean casting

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly, but boolean cannot. We say, boolean is incompatible for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.

```
boolean x = true;  
  
int y = x;           // error  
  
boolean x = true;  
  
int y = (int) x;    // error
```

byte → short → int → long → float → double

In the above statement, **left to right can be assigned implicitly(Up) casting and right to left requires explicit(Down) casting**. That is, byte can be assigned to short implicitly but short to byte requires explicit casting.

Object casting (reference casting) discusses casting between objects of different classes involved in inheritance.

Your one stop destination for all data type conversions

byte TO short int long float double char boolean

short TO	byte	int	long	float	double	char	boolean
int TO	byte	short	long	float	double	char	boolean
float TO	byte	short	int	long	double	char	boolean
double TO	byte	short	int	long	float	char	boolean
char TO	byte	short	int	long	float	double	boolean
boolean TO	byte	short	int	long	float	double	char

String and data type conversions

String TO	byte	short	int	long	float	double	char	boolean
byte	short	int	long	float	double	char	boolean	TO String

Casting incompatible types in java

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int.

This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form: (target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte.

If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;

// ...

b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components.

Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.

class Conversion {

    public static void main(String args[]) {

        byte b;

        int i = 257;

        double d = 323.142;

        System.out.println("\nConversion of int to byte.");

        b = (byte) i;

        System.out.println("i and b " + i + " " + b);

        System.out.println("\nConversion of double to int.");

        i = (int) d;

        System.out.println("d and i " + d + " " + i);

        System.out.println("\nConversion of double to byte.");

        b = (byte) d;

        System.out.println("d and b " + d + " " + b);

    }
}
```

```
}
```

Output:

Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67

Automatic type casting / promotion in expression

There is one more place where certain type conversions may occur: In expressions.

In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.

For example, examine the following expression:

```
byte a = 40;
```

```
byte b = 50;
```

```
byte c = 100;
```

```
int d = a * b / c; // a * b exceeds the range of byte
```

The result of the intermediate term $a * b$ easily exceeds the range of either of its byte operands.

To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression.

This means that the sub expression $a * b$ is performed using integers-not bytes.

Thus, 2,000, the result of the intermediate expression, $50 * 40$, is legal even though a and b are both specified as type byte.

As useful as the automatic promotions are, they can cause confusing compile-time errors.

For example, this seemingly correct code causes a problem:

```
byte b = 50;  
b = b * 2; // Compile time error, Cannot assign an int to a byte!
```

The code is attempting to store $50 * 2$, a perfectly valid byte value, back into a byte variable.

However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;  
b = (byte)(b * 2); // which yields the correct value of 100.
```

Wrapper classes

Introduction

Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with `java.io.File`), an address of a system can be seen as an object (with `java.util.URL`), an image can be treated as an object (with `java.awt.Image`) and a simple data type can be converted into an object (with wrapper classes). This tutorial discusses wrapper classes. Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, upto JDK1.4, the data structures accept only

objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced wrapper classes.

What are Wrapper classes:

As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it.

Observe the following conversion.

```
int k = 100;  
  
Integer it1 = new Integer(k);
```

The int data type k is converted into an object, it1 using Integer class. The it1 object can be used in Java programming wherever k is required an object.

The following code can be used to unwrap (getting back int from Integer object) the object it1.

```
int m = it1.intValue();  
  
System.out.println(m*m); // prints 10000
```

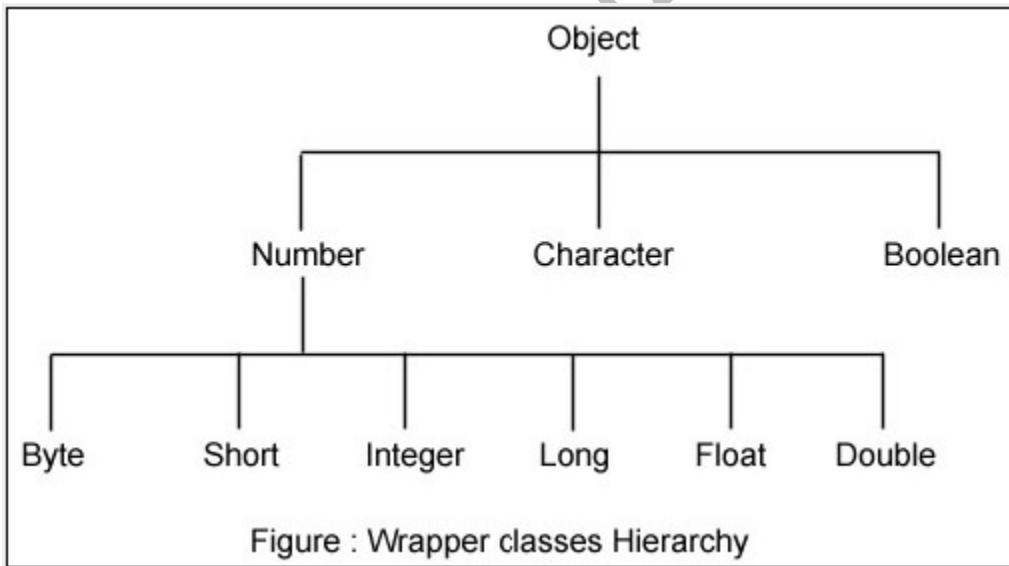
intValue() is a method of Integer class that returns an int data type.

List of Wrapper classes

In the above code, Integer class is known as a wrapper class (because it wraps around int data type to give it an impression of object). To wrap (or to convert) each primitive data type, there comes a wrapper class. Eight wrapper classes exist in java.lang package that represent 8 data types. Following list gives.

Primitive	Wrapper	Class Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String

char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String



All the 8 wrapper classes are placed in `java.lang` package so that they are implicitly imported and made available to the programmer. As you can observe in the above hierarchy, the super class of all numeric wrapper classes is `Number` and the super class for `Character` and `Boolean` is `Object`. All the wrapper classes are defined as final and thus designers prevented them from inheritance.

Importance of Wrapper classes

There are mainly two uses with wrapper classes.

To convert simple data types into objects, that is, to give object form to a data type; here constructors are used.

To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.

The following program expresses the style of converting data type into an object and at the same time retrieving the data type from the object.

Exp:-

```
public class WrappingUnwrapping
{
    public static void main(String args[])
    {
        byte grade = 2;                                // data types
        int marks = 50;
        float price = 8.6f;                            // observe a suffix of <strong>f</strong> for float
        double rate = 50.5;                            // data types to objects
        Byte g1 = new Byte(grade);                     // wrapping
        Integer m1 = new Integer(marks);
        Float f1 = new Float(price);
        Double r1 = new Double(rate);
        // let us print the values from objects
        System.out.println("Values of Wrapper objects (printing as objects)");
        System.out.println("Byte object g1: " + g1);
```

```
System.out.println("Integer object m1: " + m1);
System.out.println("Float object f1: " + f1);
System.out.println("Double object r1: " + r1);

        // objects to data types (retrieving data types from objects)

byte bv = g1.byteValue();           // unwrapping

int iv = m1.intValue();

float fv = f1.floatValue();

double dv = r1.doubleValue();

        // let us print the values from data types

System.out.println("Unwrapped values (printing as data types)");

System.out.println("byte value, bv: " + bv);

System.out.println("int value, iv: " + iv);

System.out.println("float value, fv: " + fv);

System.out.println("double value, dv: " + dv);

}

}
```

Output:

Values of Wrapper objects (printing as objects)

Byte object g1: 2

Integer object m1: 50

Float object f1: 8.6

Double object r1: 50.5

Unwrapped values (printing as data types)

byte value, bv: 2

int value, iv: 50

float value, fv: 8.6

double value, dv: 50.5

Exp:-

```
public class FindRangesWrapper
{
    public static void main(String[] args)
    {
        System.out.println("Integer range:");
        System.out.println(" min: " + Integer.MIN_VALUE);
        System.out.println(" max: " + Integer.MAX_VALUE);

        System.out.println("Double range:");
        System.out.println(" min: " + Double.MIN_VALUE);
        System.out.println(" max: " + Double.MAX_VALUE);

        System.out.println("Long range:");
        System.out.println(" min: " + Long.MIN_VALUE);
        System.out.println(" max: " + Long.MAX_VALUE);

        System.out.println("Short range:");
        System.out.println(" min: " + Short.MIN_VALUE);
        System.out.println(" max: " + Short.MAX_VALUE);

        System.out.println("Byte range:");
        System.out.println(" min: " + Byte.MIN_VALUE);
        System.out.println(" max: " + Byte.MAX_VALUE);

        System.out.println("Float range:");
    }
}
```

```
System.out.println(" min: " + Float.MIN_VALUE);
System.out.println(" max: " + Float.MAX_VALUE);
}
```

Output:

Integer range:

min: -2147483648

max: 2147483647

Double range:

min: 4.9E-324

max: 1.7976931348623157E308

Long range:

min: -9223372036854775808

max: 9223372036854775807

Short range:

min: -32768

max: 32767

Byte range:

min: -128

max: 127

Float range:

min: 1.4E-45

max: 3.4028235E38

AutoBoxing and AutoUnBoxing:

Until Java 1.4 version we are not allowed to provide primitive value in the place of Wrapper object in the place of primitive directly.

Programmer is responsible to convert primitive to wrapper object, Wrapper object to primitive based on the requirement.

From Java 5.0 version onwards we can provide Wrapper object in the place of primitive and primitive in the place of Wrapper object directly.

The required conversion automatically performed by the compiler that is nothing but AutoBoxing and AutoUnBoxing.

AutoBoxing:

Automatic conversion from primitive to Wrapper object by the compiler is called AutoBoxing.

Exp:-

```
class Auto
{
    public static void main(String[] args)
    {
        int x=38;
        Integer i=x; //AutoBoxing
        System.out.println(i);
    }
}
```

AutoUnBoxing:

Automatic conversion from Wrapper object to primitive by the compiler is called AutoUnBoxing.

Exp:-

```
class UnBoxing
{
    public static void main(String[] args)
    {
        Integer i=new Integer(40);
        int x=i;//AutoUnBoxing
        System.out.println(x);
    }
}
```

COREJAVA BY NAGARJUNA

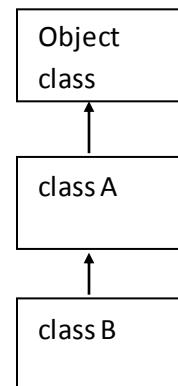
Chapter:3 Object Class

toString()
hashcode()
equals()
finalize()
clone()
final getClass()
final wait()
final wait(long timeout)
final wait(long timeout, int nanos)
final notify()
final notifyAll

Object class present in Java.lang package every java class, user defined class (or) predefined class is the sub classes of object class directly (or) indirectly.

Exp:-

```
class A  
{  
}  
  
class B extends A  
{  
}
```



Object class as act has a root class of java tech.

Object class contains following 11 methods.

- 1) `toString()`
- 2) `hashCode()`
- 3) `equals()`
- 4) `finalize()`
- 5) `clone()`
- 6) `final getClass()`
- 7) `final wait()`
- 8) `final wait(long timeout)`
- 9) `final wait(long timeout, int nanos)`
- 10) `final notify()`
- 11) `final notifyAll()`

public final Class getClass(): runtime objet class delimitation can be return and

public int hashCode(): returns the hashcode number for this object.

public boolean equals(Object obj): compares the given object to this object.

protected Object clone() throws CloneNotSupportedException

creates and returns the exact copy (clone) of this object.

public String toString(): returns the string representation of this object.

public final void notify(): wakes up single thread, waiting on this object's monitor.

public final void notifyAll(): wakes up all the threads, waiting on this object's monitor.

public final void wait(long timeout) throws InterruptedException

This method causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

public final void wait(long timeout, int nanos) throws InterruptedException

This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

public final void wait() throws InterruptedException

This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

protected void finalize() throws Throwable

Is invoked by the garbage collector before object is being garbage collected.

Q) What are the methods we can override in sub class from object class?

We can override the five methods based on application requirement because they are non-final methods in the object class.

- 1) hashCode()
- 2) equals()
- 3) toString()
- 4) clone()
- 5) finalize()

If we are not overriding the object class methods in the sub class there is no compile time and run time exception.

toString:

This method returns string representation of an object, if we are bypassing any object reference variables directly to S.O.Pln statement JVM will invoke `toString()` method.

The object class `toString` method returns always “class name@ hexadecimal format of hashCode”.

Object class `toString()` method is implemented as follows.

internal code:

```
class
{
    public String toString()
    {
        return getClass().getName()+" "+Integer.toHexString(hashCode());
    }
}
```

Advantage of the `toString()` method:

By overriding the `toString()` method of the Object class, we can return values of the object, so we don't need to write much code.

Exp:-

```
class Student
{
    int rollno;
    String name;
    String city;
    Student(int rollno, String name, String city){
        this.rollno=rollno;
```

```
        this.name=name;  
  
        this.city=city;  
    }  
  
    public static void main(String args[]){  
  
        Student s1=new Student(101,"Raj","usa");  
  
        Student s2=new Student(102,"Nag","india");  
  
        System.out.println(s1);//compiler writes here s1.toString()  
  
        System.out.println(s2);//compiler writes here s2.toString()  
    }  
}
```

As we can see in the above example, printing s1 and s2 prints the hashCode values of the objects but I want to print the values of these objects. Since java compiler internally calls `toString()` method, overriding this method will return the specified values. Let's understand it with the example given below:

Exp:-

```
class Student{  
  
    int rollno;  
  
    String name;  
  
    String city;  
  
    Student(int rollno, String name, String city){  
  
        this.rollno=rollno;  
  
        this.name=name;  
  
        this.city=city;  
    }  
  
    public String toString(){//overriding the toString() method
```

```
        return rollno+" "+name+" "+city;  
    }  
  
    public static void main(String args[]){  
        Student s1=new Student(101,"Raj","usa");  
        Student s2=new Student(102,"nag","india");  
        System.out.println(s1);//compiler writes here s1.toString()  
        System.out.println(s2);//compiler writes here s2.toString()  
    }  
}  
  
public int hashCode();
```

Syntax: public native int hashCode()

To get the hashCode of the object we can call hashCode() method.

We can also override the hashCode method in user defined classes. to create our own hash codes by using object state.

JVM will always uses the hashCode to save the object into hash table, hash map and hash set.

By following ways we can override the hash code.

Exp:-

```
public int hashCode()  
{  
    return 100; // improper way type  
}
```

```
public int hashCode()
```

```
{  
    return rollno; // valid  
}
```

Exp:-

```
public class TestEmp  
{  
    private int age ;  
    public TestEmp( int age )  
    {  
        this.age = age;  
    }  
    public int hashCode()  
    {  
        return age;  
    }  
    public static void main(String[] args)  
    {  
        TestEmp emp1 = new TestEmp(23);  
        System.out.println("Overridden hashCode()--->>>" + emp1.hashCode());  
        int originalHashCode = System.identityHashCode(emp1);  
        System.out.println("Original hashCode of Emp--->>>" + originalHashCode);  
    }  
}
```

Output:

```
D:\>javac TestEmp.java
```

```
D:\>java TestEmp
```

Overridden hashCode()--->>>23

Original hashCode of Emp---->>>11352996

public boolean equals():

equals method is used to compare the two objects. In java we can compare object in two ways.

- 1) by using object reference
- 2) by using object state

If two objects of a class are said to be equal only if their reference are state should is equal.

We have to compare the objects either by using “==” operator it always compares the objects with their reference.

By using the equal's method we can compare the object based on its implementation.

Object class equals method by default compares the object reference. By overriding the equal's method in the user defined class we can compare the object with their state.

Object class equals method internal implementation:

Object class equals method implemented as follows.

Exp:-

```
class Object
{
    public boolean equals(Object obj)
    {
        return (this==obj)
    }
}
```

```
}
```

This method returns true if the current object reference is equals to argument object reference otherwise it returns false.

Exp:-

```
class StudentEquals{  
    int rollno;  
    String name;  
    String city;  
    StudentEquals(int rollno, String name, String city){  
        this.rollno=rollno;  
        this.name=name;  
        this.city=city;  
    }  
    public static void main(String args[]){  
        Student s1=new Student(101,"nag","ind");  
        Student s2=new Student(101,"arjun","usa");  
        Student s3=s1;  
        System.out.println(s1==s2);//false  
        System.out.println(s1.equals(s2));//false  
        System.out.println(s1==s3);//true  
        System.out.println(s1.equals(s3));//true  
    }  
}
```

Overriding the equals() method:

Exp:-

```
public class EmpEquals
{
    private int age ;
    public EmpEquals( int age )
    {
        super();
        this.age = age;
    }

    public int hashCode()
    {
        return age;
    }

    public boolean equals( Object obj )
    {
        boolean flag = false;
        EmpEquals emp = ( EmpEquals )obj;
        if( emp.age == age )
            flag = true;
        return flag;
    }

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

```
{  
    EmpEquals emp1 = new EmpEquals(23);  
    EmpEquals emp2 = new EmpEquals(23);  
    System.out.println("emp1.equals(emp2)--->>" + emp1.equals(emp2));  
}  
}
```

Output:

True

Contract b/w hashCode() and equals() methods:

If we are overriding the equals() method then it is always recommended to override the hashCode method because if the equals method returns true by comparing two objects then hashCode of both objects must be same. If equals method return false the hashCode of both objects may or may not be same.

Clone ():

Clone is nothing but the process of copying one object to produce the exact object, we cannot copy one object directly to another object. So we have cloning process to achieve this objective.

Object cloning means creating the duplicate copy with the current object state.

Object cloning creates a new object means original object and a cloned object has different references. But both are having the same state , hence they are different objects that's why if we are performed any modification on one object that will not affect the other object .

Requirement: If you want to create the second object using the first object current modified state we must use the cloning.

If we are using new operator and constructor always objects are created with default state.

To create duplicate object we must be use object class clone () method.

Note:

To perform cloning operation an object of that class must be implement the Cloneable interface. It is a marker interface it provides permission to JVM to clone a object.

If the class is not implemented Cloneable interface then JVM will raise an exception **java.lang.CloneNotSupportedException**.

Types of cloning:

Java tech supports two types of cloning.

- 1) Shallow cloning.
- 2) Deep cloning.

How will you achieve cloning in java:

In Java everything is achieved through class, object and interface .By default no Java class support cloning but Java provide one interface called Cloneable, which is a and by implementing this interface we can make duplicate copy of our object by calling `clone()` method of `java.lang.Object` class. This Method is protected inside the object class and Cloneable interface is a marker interface and this method also throw **CloneNotSupportedException** if we have not implement this interface and try to call `clone()` method of Object class. By default anyclone() method gives **shallow copy** of the object i.e. if we invoke `super.clone()` then it's a shallow copy but if we want to **deep copy** we have to override the `clone()` method and make it public and give own definition of making copy of object. Now we let's see what is shallow and deep copy of object in Java programming language.

Shallow Copy:

Whenever we use default implementation of `clone` method we get shallow copy of object means it create new instance and copy all the field of object to that new instance and return it as **object type** we need to explicitly cast it back to our original object. This is shallow copy of the object. `clone()` method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable In shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves. That's why the name shallow copy or shallow cloning in Java. If only primitive type fields or are there then there is no difference between shallow and deep copy in Java.

Deep Copy:

Whenever we need own meaning of copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need. So for deep copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class. After that we override the clone() method in all those classes even in the classes where we have only primitive type members otherwise we would not be able to call the protected clone() method of Object class on the instances of those classes inside some other class. It's typical restriction of the protected access.



Difference between Shallow and Deep Copy in Java:

I think now we know what is deep and shallow copy of object in Java, let see some difference between them so that we can get some more clarity on them.

- When we call Object.clone(), this method performs a shallow copy of object, by copying data field by field, and if we override this method and by convention first call super.clone(), and then modify some fields to "deep" copy, then we get deep copy of object. This modification is done to ensure that original and cloned object are independent to each other.
- In shallow copy main or parent object is copied, but they share same fields or children if fields are modified in one parent object other parent fields have automatic same changes occur, but in deep copy this is not the case.
- If our parent object contains only primitive value then shallow copy is good for making clone of any object because in new object value is copied but if parent object contains any other object then only reference value is copied in new parent object and both will point to same object so in that case according to our need we can go for deep copy.
- Deep copy is expensive as compare to shallow copy in terms of object creation, because it involves recursive copying of data from other mutable objects, which is part of original object.

This is all about deep copy and shallow copy of objects in Java. Now the question comes when we use shallow copy and when go for deep copy , so answer would be simple that if the object has only primitive fields or Immutable objects, then obviously we will go for shallow copy, but if the object has references to other mutable objects, then based on the requirement, shallow copy or deep copy can be chosen.

Means if the references are not modified anytime, then there is no point in going for deep copy, We can go for shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

finalize ():

Syn:- protected void finalize()

finalize method is automatically called and executed by garbage collector to maintain the cleanup activities when an object is destroyed. We can also call the finalize method explicitly but it is not recommended.

Inside in object class finalize method is having an empty implementation, because clean up code are recourse releasing logic is specific to subclasses.

getClass ():

public final java.lang.class getClass()

getClass() method returns the runtime class object definition.

Exp:-

```
class TestExp
{
    public static void main(String[] args)
    {
        TestExp te=new TestExp();
        System.out.println(te.getClass());
    }
}
```

Output: class TestExp

Chapter4: Strings

String

String Buffer

String Builder

StringTokenizer

COREJAVA BY NAGARJUNA

String:

String is a sequence of characters into double quotes(" "). To perform the string operations sun micro system has provided the following three classes in java. Lang package.

- String
- String Buffer
- String Builder

String:

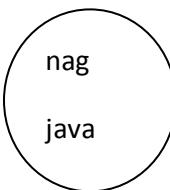
String class object are immutable objects that once we created a string object we can't perform any changes in the existing object. If we are trying to perform any changes with exacting object a new string object will be created. This behavior is nothing is but immutability of a string object.

Exp:-

```
class StringDemo
{
    public static void main(String[] args)
    {
        String s=new String("nag");
        s.concat("arjun");
        s.concat("java");
        System.out.println(s);
    }
}
```



SCP Area



both are created in heap

not having any reference.

Output: nag

String class constructors:

1) String():

```
String s=new String();
```

This constructor is used to create an empty string object.

Exp:-

```
class StringDemo
{
    public static void main(String[] args)
    {
        String s=new String();
        System.out.println(s);
    }
}
```

Output: Space

2) String(String s):

```
String s=new String("nag");
```

This constructor is used to create a string object with the provided content

Exp:-

```
class StringDemo
{
    public static void main(String[] args)
    {
        String s=new String("nag");
        System.out.println(s);
    }
}
```

```
}
```

```
}
```

Output: nag

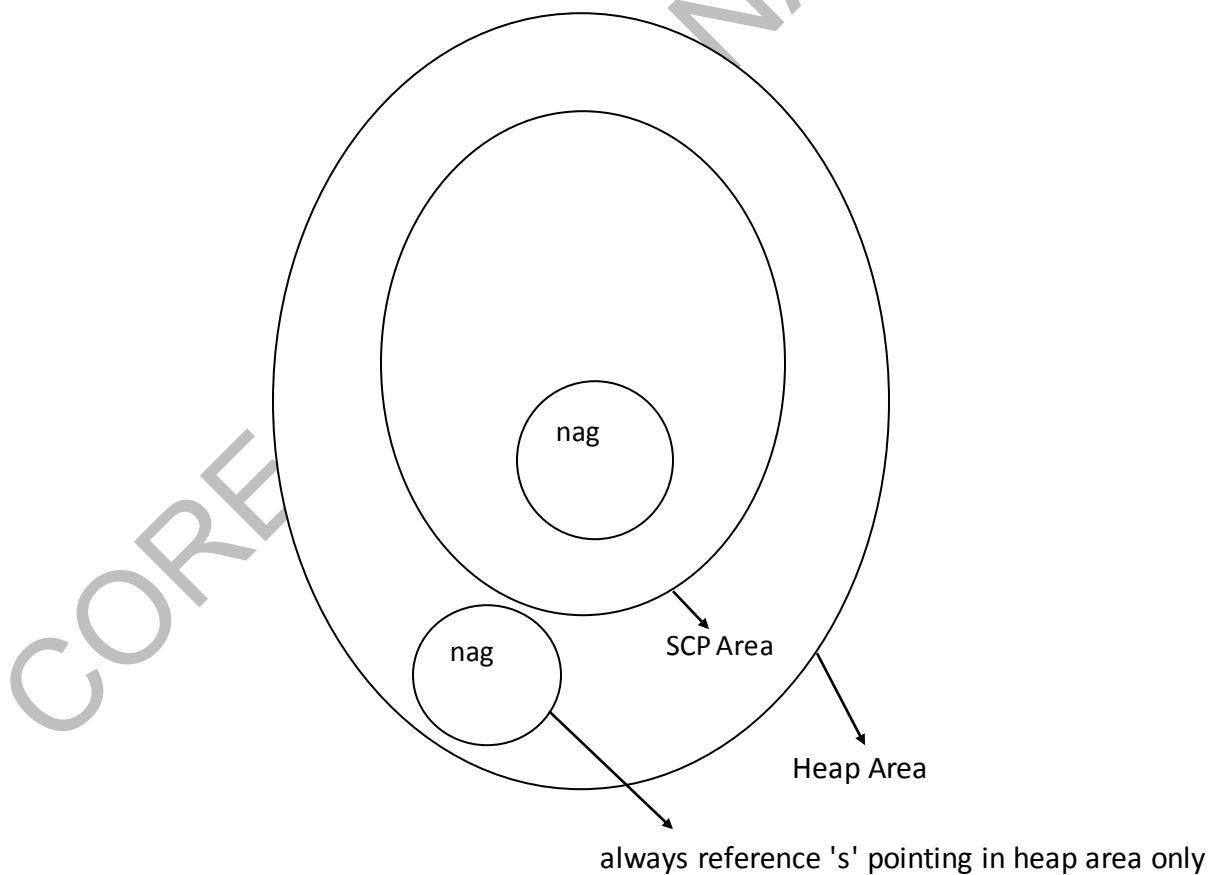
Q) what is difference b/w following two statements?

```
String s=new String("nag");
```

In this case two string objects are created that is one is in heap area and another one is **SCP** (string constant pool area), and 's' is always reference heap area object only.

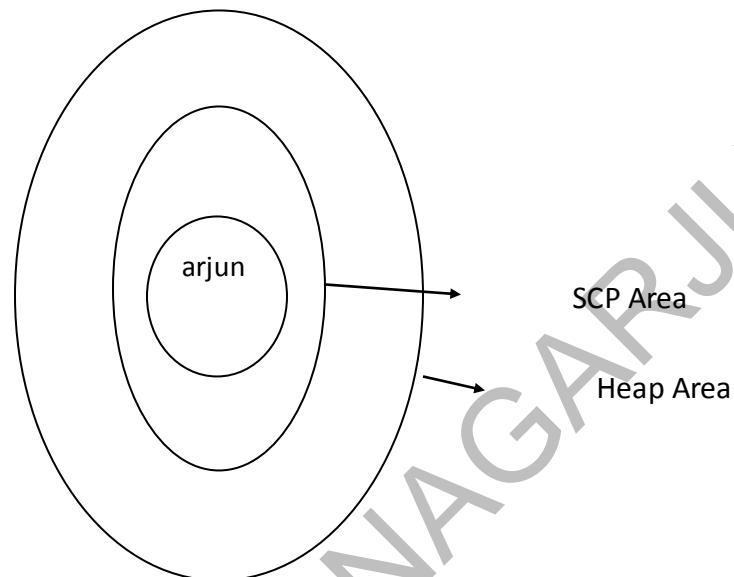
In **SCP** area that object does not contains any reference even though garbage collector is not allowed to destroy the object. Because garbage collector not allowed to destroy the objects in **SCP** area.

SCP area objects are automatically destroyed when ever JVM was shutdown.



String s="arjun":

In this case only one object will be created in **SCP** area and always 's' refers SCP area only.

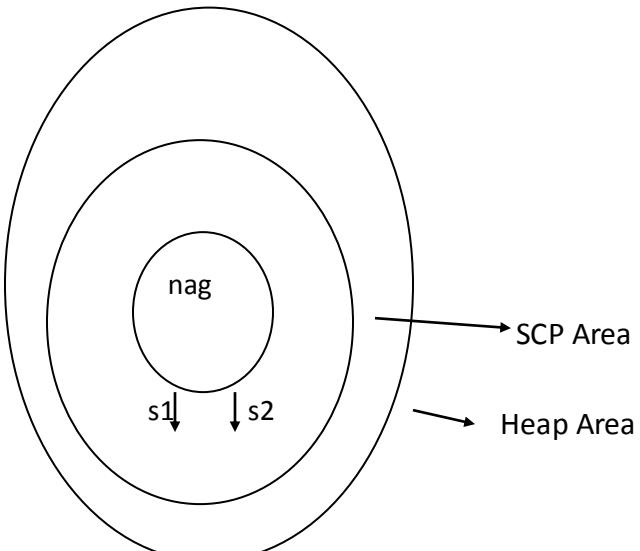


Exp:-

```
String s1="nag";
```

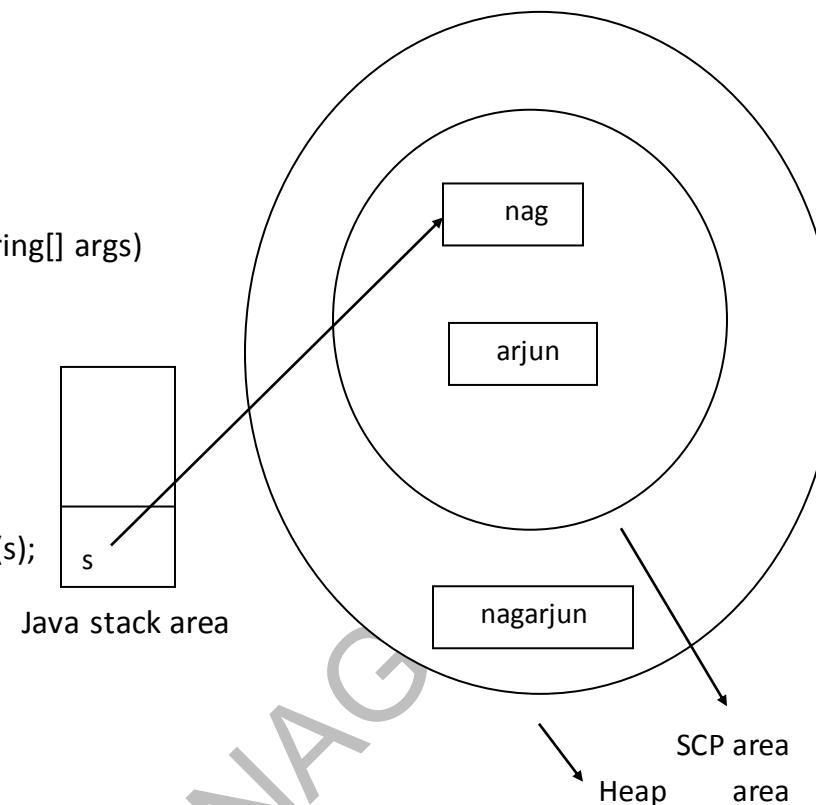
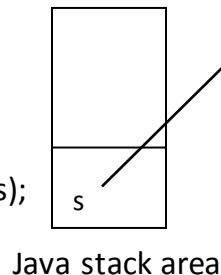
```
String s2="nag"; // no new object will be created.
```

Whenever we are creating a string object with the literal the JVM will checks the string constant pool first, if the string is already existed in the pool area a reference to the pooled instance returns if the string doesn't existing in the pool area a new string object will be created and placed in the pool area.



Exp:-

```
class StringDemo
{
    public static void main(String[] args)
    {
        String s="nag";
        s.concat("arjun");
        System.out.println(s);
    }
}
```



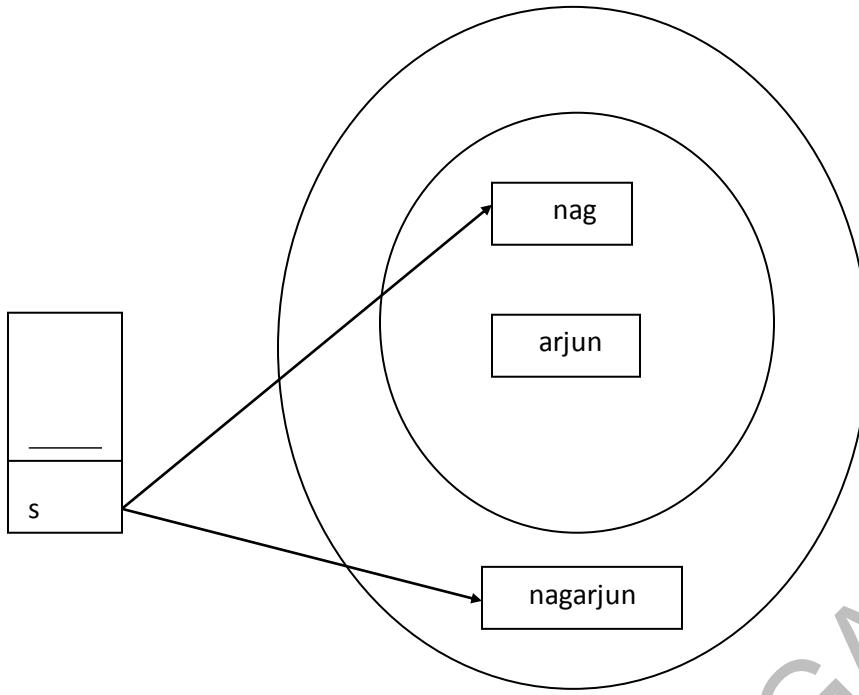
OUTPUT: nag

In the above diagram total three object copies are created two in SCP area and one in Heap area, No reference variable refers to heap area object "nagarjun" and no reference is refer "arjun" in SCP area.

Exp:-

```
class StringDemo
{
    public static void main(String[] args)
    {
        String s="nag";
        s=s.concat("arjun");
        System.out.println(s);
    }
}

//OUTPUT: nagarjun
```



Importance of SCP area:

In our program if the string object is repeatedly created then it is never recommended to create a separate object (using new operator) for every requirement with the same content. It reduces the memory utilization and create the performance problems.

We can create only one copy in SCP area (using literal) and reuse the same object for every requirement. This approach improves the performance of the system and memory utilization this is the advantage of SCP.

suppose a several reference variables are pointing into the same object by using one reference we can perform any changes value of the object it will be affected to all the reference variables. That is why string objects are immutable in java.

The main advantage of SCP area is performance improvement.

String(byte[]b):

This constructor can be used to create a string class object with the string equivalent of specified byte array content.

Exp:-

```
class ByteArray
```

```
{  
    public static void main(String[] args)  
    {  
        byte[] b={98,97,99,100,101};  
        String s=new String(b);  
        //System.out.println(b);//classname@hashcode  
        System.out.println(s);//12345  
    }  
}
```

String(byte[]b, int string index, int):

This constructor can be used to create string class objects with the string equivalent specified array content start from the specified start index level and upto specified no of elements.

Exp:-

```
class ByteArray  
{  
    public static void main(String[] args)  
    {  
        byte[] b={98,97,99,100,101};  
        String s=new String(b,1,2);  
        System.out.println(s);//ac  
        String s1=new String(b,0,3);  
        System.out.println(s1);//bac  
    }  
}
```

```
}
```

String(char[] ch):

This constructor is used for create a new string object based on the provided array content.

Exp:-

```
class CharArray  
{  
    public static void main(String[] args)  
    {  
        char[] b={'c','d','y','a','n'};  
        String s=new String(b);  
        System.out.println(s);//cdyan  
    }  
}
```

String(StringBuffer sb):

String(StringBuilder sb):

Importance Methods in String class:

1) Concat(String s):

Syn:- public String Concat(String s)

There are two ways to concat string objects:

1. By + (string concatenation) operator
2. By concat() method

1) By + (string concatenation) operator:

String concatenation operator is used to add strings.

For Example:

```
//Example of string concatenation operator  
class Simple{  
  
public static void main(String args[]){  
  
String s="Sachin"+" Tendulkar";  
  
System.out.println(s);//Sachin Tendulkar  
  
}  
  
}
```

Output:Sachin Tendulkar

The compiler transforms this to:

```
String s=(new StringBuilder()).append("Sachin").append(" Tendulkar").toString();
```

String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its `append` method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

```
class Simple{  
  
public static void main(String args[]){  
  
String s=50+30+"Sachin"+40+40;  
  
System.out.println(s);//80Sachin4040  
  
}  
  
}
```

Output:80Sachin4040

Note:If either operand is a string, the resulting operation will be string concatenation. If both operands are numbers, the operator will perform an addition.

2. By concat() method:

Concat method is used for add some content to the existing string object

Concat method always returns a new string object.

Exp:-

```
class Concat
{
    public static void main(String[] args)
    {
        String s="nag";
        String s1=s.concat("arjun");
        String s2=s1.concat("raj");
        System.out.println(s);//nag
        System.out.println(s1);//nagarjun
        System.out.println(s2);//nagarjunraj
    }
}
```

equals(object obj):

String comparison:

There are three ways to compare String objects:

1. By equals() method
2. By == operator
3. By compareTo() method

1) By equals() method:

equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- public boolean equals(Object another){} compares this string to the specified object.
- public boolean equalsIgnoreCase(String another){} compares this String to another String, ignoring case

Example of equals(Object) method:-

```
class Simple{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        String s4="Saurav";  
        System.out.println(s1.equals(s2));//true  
        System.out.println(s1.equals(s3));//true  
        System.out.println(s1.equals(s4));//false  
    }  
}
```

Output:

true

true

false

Example of equalsIgnoreCase(String) method:-

```
class Simple{
```

```
public static void main(String args[]){
    String s1="Sachin";
    String s2="SACHIN";
    System.out.println(s1.equals(s2));//false
    System.out.println(s1.equalsIgnoreCase(s2));//true
}
```

Output:

false

true

2) By == operator:

The == operator compares references not values.

Example of == operator:-

```
class Simple{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="Sachin";
        String s3=new String("Sachin");
        System.out.println(s1==s2);//true (because both refer to same instance)
        System.out.println(s1==s3);//false(because s3 refers to instance created in
nonpool)
    }
}
```

Output:

true

false

3) By compareTo() method:

compareTo() method compares values and returns an int which tells if the values compare less than, equal, or greater than.

Suppose s1 and s2 are two string variables.If:

s1 == s2 :0

s1 > s2 :positive value

s1 < s2 :negative value //Example of compareTo() method:

Exp:-

```
class Simple
{
    public static void main(String args[])
    {
        String s1="Sachin";
        String s2="Sachin";
        String s3="Ratan";
        System.out.println(s1.compareTo(s2));
        //0 System.out.println(s1.compareTo(s3));//1(because s1>s3)
        System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
    }
}
```

Output:

0

1

-1

split(String s):

The split method divided the given string into the number of tokens based on the specified delimiter and returns string array.

Exp:-

```
class Split  
{  
    public static void main(String[] args)  
    {  
        String s="welcome to My company";  
        String[] res=s.split(" ");  
        for(int i=0;i<res.length;i++)  
        {  
            System.out.println(res[i]);  
        }  
    }  
}
```

Output:

welcome
to
My
company

Q) Write java program reverse the string words?

```
class Reverse
{
    public static void main(String[] args)
    {
        String original="Hi Hello HRU";
        String[] s=original.split(" ");
        String res="";
        for(int i=s.length-1;i>=0;i--)
        {
            res=res+s[i]+" ";
        }
        System.out.println("original string:"+original);
        System.out.println("reverse string:"+res);
    }
}
```

Output:

original string: Hi Hello HRU

reverse string: HRU Hello Hi

Q) If we want to output like original URH olleH iH just remove the space above program following two lines.

```
String[] s=original.split("");
res=res+s[i]+"";
```

Output:

string: Hi Hello HRU

reverse string: URH olleH iH

byte[]getbytes():

This method returns an equivalent byte array for the given string object.

Exp:-

```
class CharToByte
{
    public static void main(String[] args)
    {
        String s="Hi Hello HRU";
        byte[] b =s.getBytes();
        for(int i=0;i<b.length;i++)
        {
            System.out.println((char) b[i]"--"+b[i]);
        }
    }
}
```

Output:

H-72

i-105

--32

H-72

e-101

I--108

I--108

O--111

--32

H--72

R--82

U--85

String trim():

trim method removes starting and ending spaces of the given string, but not the middle spaces. and returns a new string object.

Exp:-

```
class Trim  
{  
    public static void main(String[] args)  
    {  
        String s=" Hi Hello HRU  ";  
        System.out.println(s);  
        System.out.println(s.length());  
        String s1=s.trim();  
        System.out.println(s1);  
        System.out.println(s1.length());  
    }  
}
```

Output:

Hi Hello HRU

16

Hi Hello HRU

12

String subString(int):

String subString(int,int):

String subString(int starting index):

substring method returns a new string from specified starting index position on words.

String subString(int starting index, end index):

This index returns a new string from specified starting index position and specified end -1 index position.

Exp:-

```
class SubString
{
    public static void main(String[] args)
    {
        String s="HiHelloHRU";
        String s1=s.substring(3);
        String s2=s.substring(7);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output:

HelloHRU

HRU

Exp:-

```
class SubString
{
    public static void main(String[] args)
    {
        String s="HiHelloHRU";
        String s3=s.substring(1,3);
        String s4=s.substring(4,8);
        System.out.println(s3);
        System.out.println(s4);
    }
}
```

Output:

iH

IIOH

char charAt(int index):

charAt method returns character based on the specified index position. If the specified index is not available then JVM will raise the following exception.

Exp:-

```
class DeleteCharAt
{
```

```
public static void main(String[] args)
{
    StringBuffer sb=new StringBuffer();
    sb.append("hi hello hru");
    System.out.println(sb);
    sb.deleteCharAt(2);
    System.out.println(sb);
    sb.deleteCharAt(8);
    System.out.println(sb);
    sb.deleteCharAt(10);//
    System.out.println(sb);SIOOBE
}
}
```

Output:

hi hello hru

hihello hru

hihello ru

String Buffer:

String Buffer objects are mutable objects. Once we created a string buffer object we can perform any changes in the existing string buffer object is allowed. This behavior is nothing but mutable of string buffer object.

Exp:-

```
class TStringBuffer
{
```

```
public static void main(String[] args)
{
    StringBuffer sb=new StringBuffer("nag");
    sb.append("arjun");
    sb.append("good boy");
    System.out.println(sb);
}
```

Output:-

nagarjunagood boy

StringBuffer constructors:

String Buffer():

This is zero argument constructor

This constructor creates an empty String Buffer object with the default initial capacity is 16. (length is 0).

If the String Buffer object reaches its max capacity then a new string buffer object will be created with the capacity as

Exp:-

(currentcapacity*2)+2

new capacity (16*2)+2

new capacity= 34

Exp:-

```
class StringBuffer2
```

```
{
```

```
public static void main(String[] args)
{
    StringBuffer sb=new StringBuffer();
    System.out.println(sb.capacity());//16
    System.out.println(sb.length());//0
    sb.append("abcajhgahgfhagfhagfhgaf");
    System.out.println(sb.capacity());//34
    System.out.println(sb.length());//23
    sb.append("123456789");
    System.out.println(sb.capacity());//34
    System.out.println(sb.length());//32
}
}
```

StringBuffer(int capacity):

This constructor creates a new string buffer object based on the provided capacity.

Exp:-

```
StringBuffer sb=new StringBuffer(90);
s.o.println(s.capacity()) //90
s.o.println(s.length()) //0
```

StringBuffer(String s):

This constructor creates a new string buffer object based on provided string content. This constructor is used for converting string object into string buffer object.

here string buffer capacity is =default capacity of sb + length of string

Exp:-

```
class StringBuffer3
{
    public static void main(String[] args)
    {
        String s="abcde";
        StringBuffer sb=new StringBuffer(s);
        System.out.println(sb.capacity());//21
        System.out.println(sb.length());//5
        sb.append("abcajhgahgfhagfhagfn");//20
        System.out.println(sb.capacity());//44=(21*2)+2
        System.out.println(sb.length());//25(5+20)
        -
    }
}
```

Important methods in String Buffer:

append():

This method is overloaded method in string buffer class this method is used for to add XXX (string, object) data at the end of source to string buffer object.

Capacity():

```
public int capacity()
```

This method is used for determining the capacity of string buffer object.

length():

```
public int length()
```

This method is used for determining the length of the string buffer object.

reverse():

```
public string buffer reverse()
```

This method is used for reverse the content of string buffer object.

Exp:-

```
class Reverse1
{
    public static void main(String[] args)
    {
        String s="hi hello nag";
        StringBuffer sb=new StringBuffer(s);
        System.out.println(sb);

        sb.reverse();
        System.out.println(sb);
    }
}
```

Output:

```
hi hello nag
gan olleh ih
```

deletecharAt(int index):

```
public StringBuffer deleteCharAt(int index)
```

This method is deleted a particular character based on the specified index.

Exp:-

```
class StringBufferExp
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer();
        sb.append("hi hello hru");
        System.out.println(sb);
        sb.deleteCharAt(2);
        System.out.println(sb);
        sb.deleteCharAt(8);
        System.out.println(sb);
        //sb.deleteCharAt(10);
        //System.out.println(sb); //SIOOBE
    }
}
```

OutPut:

hi hello hru

hihello hru

hihello ru

StringBuffer delete(int, int):

```
public StringBuffer delete(int,int)
```

This method is used for removing the specified number of characters from specified starting index value to specified end -1 index values.

If the specified starting index is not available then JVM will raise the following error like

Java.lang.StringIndexOutOfBoundsException

If the specified ending index is not available the JVM will not raise any exception.

Exp:-

```
class Delete
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer();
        sb.append("welcome to hyd");
        sb.delete(2,4);//weome to hyd
        sb.delete(2,4);//wee to hyd
        sb.delete(1,6);//w hyd (it will take wee to hyd and apply the delete
                      //method)
        System.out.println(sb);
    }
}
```

String Builder:

StringBuilder class was introduced in java5.0 version.

StringBuilder class is similar to StringBuffer except the following difference.

String Buffer

- 1) All the methods are synchronized
- 2) Performance is low
- 3) It is thread safe
- 4) It is introduced in java1.0

String Builder

- 1) All methods are non synchronized
- 2) Performance is high
- 3) It is not a thread safe
- 4)It is introduced in java5.0

String Tokenizer: Java.util package

StringTokenizer class present in Java.util package

The main advantage of StringTokenizer is to divide the given input string into the number of tokens.

StringTokenizer can also be utilized in application to count number of words in the text file and number of time a particular word is repeated in a text file.

Exp:-

```
import java.util.StringTokenizer;  
  
class StringTokenizerExmp  
{  
  
    public static void main(String[] args)  
    {  
  
        String s="welcome to nag";  
  
        StringTokenizer st=new StringTokenizer(s);  
  
        //StringTokenizer st=new StringTokenizer(s,"a",true);  
  
        //StringTokenizer st=new StringTokenizer(s,"");  
  
        //StringTokenizer st=new StringTokenizer(s,"o");  
  
        int count=st.countTokens();  
  
        System.out.println("No Of Tokens:"+count);  
  
        while(st.hasMoreTokens())  
        {  
  
            String tokens=st.nextToken();  
  
            System.out.println(tokens);  
        }  
    }  
}
```

Constructors:

StringTokenizer(String s):

This constructor creates a StringTokenizer object with the generated tokens. StringTokenizer by using default delimiter that is space.

Exp:-

```
String s="Hi Hello";  
StringTokenizer st=new StringTokenizer(s);
```

StringTokenizer(String s, String delimiter):

This constructor creates a StringTokenizer object with the generated tokens.

Here StringTokenizer perform the tokenization on the basis of provided delimiter.

Exp:-

```
String s="Hi Hello";  
StringTokenizer st=new StringTokenizer(s,e);
```

```
import java.util.StringTokenizer;  
class StringTokenizerExmp  
{  
    public static void main(String[] args)  
    {  
        String s="welcome to nag";  
        StringTokenizer st=new StringTokenizer(s,"e");  
        //output
```

No Of Tokens:3

w

lcom

to nag

```
// here "e" is not visible

StringTokenizer st=new StringTokenizer(s,"e");

//output

No Of Tokens:5

w
e
lcom
e
to nag

// here "e" is visible

int count=st.countTokens();

System.out.println("No Of Tokens:"+count);

while(st.hasMoreTokens())

{

    String tokens=st.nextToken();

    System.out.println(tokens);

}

}
```

Methods:

public int countTokens():

This methods returns number of tokens present in StringTokenizer object.

public boolean hasMoreToken():

To check more number of tokens are available are not with respect to cursor position we should use this method.

Note: Whenever StringTokenizer object will be created before the first token JVM will created cursor automatically.

public String nextToken():

This method returns a token if the next token is available, that is hasMoreToken() returns true then this method returns nextToken.

COREJAVA BY NAGARJUNA

Chapter5: IO Streams

Stream

Byte Oriented Stream

Character oriented Stream

FileOutputStream

FileInputStream

PipedInputStream

SequenceInputStream

ObjectInputStream

ObjectOutputStream

DataOutputStream

DataInputStream

BufferedInputStream

BufferedOutputStream

PrintStream

FileWriter

FileReader

Serialization and Deserialization

Transient

BufferedReader

BufferedWriter

Byte streams and Character streams

File copying is done, in JDK 1.0, with byte streams. Byte streams can read or write the files containing ASCII characters that range from 0 to 255. That is, byte streams can copy the files containing English letters only but not of other languages.

If the file contains other than English characters, as Java supports Unicode characters, byte streams fail to do the job. To overcome this, in JDK 1.1 version, designers introduced character streams. Character streams operate on Unicode characters. That is, character streams can read, write and copy the files containing other than English characters. Characters streams can do with English

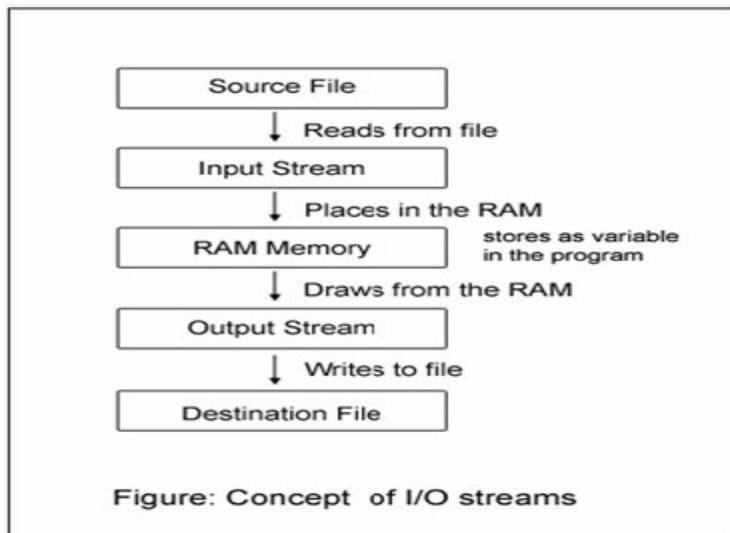
Java I/O Streams

Data can be stored on a computer system, as per the code requirements, in two ways, either permanently or temporarily. Temporary storage can be accomplished by storing the data in data structures or instance variables. The data is temporary because it is stored in RAM. For a permanent storage, the data should be stored on the hard disk either in the form of database tables or files. This tutorial is concerned with I/O streams used to write data to a file and later read from the file.

Overview of Streams

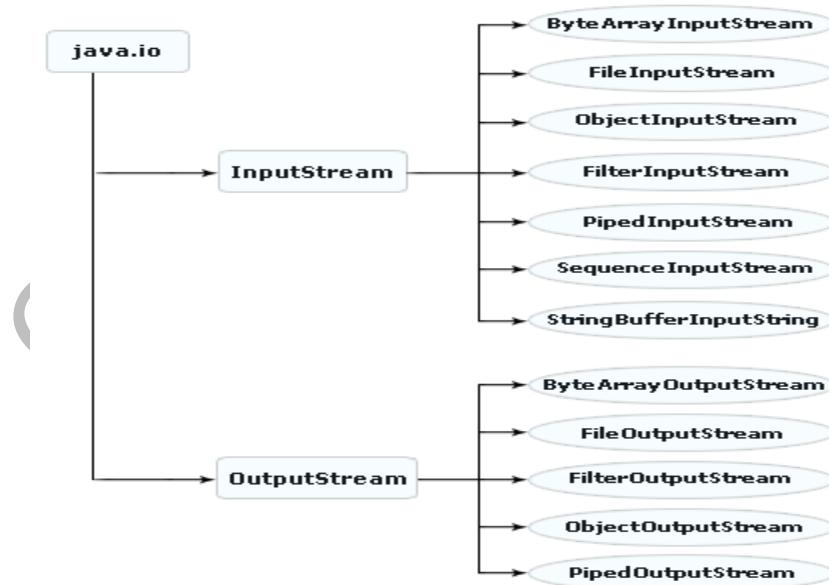
There comes two entities – a source and a destination. Source is that from where data is read and the destination is that one to where data is written. The source and destination need not be a file only; it can be a socket or keyboard input etc. To do the job of reading and writing, there comes two types of streams – input streams and output streams. An input stream job is to read from the source and the output stream job is to write to the destination. That is, in the program, it is necessary to link the input stream object to the source and the output stream object to the destination.

I/O streams are carriers of data from one place to another. The input stream carries data from the source and places it temporarily in a variable (like int k or String str etc.) in the process (program). The output stream takes the data from the variable and writes to the destination. The variable works like a temporary buffer between input stream and output stream.



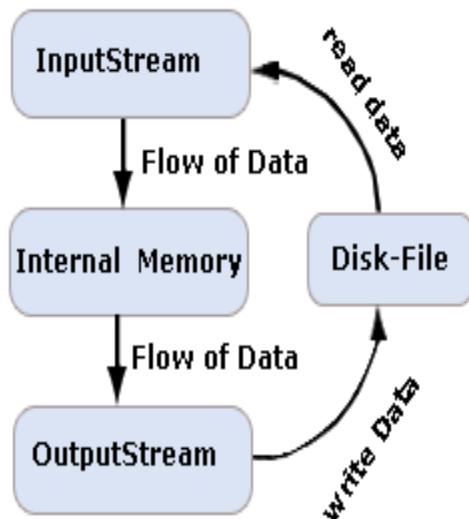
It is clear from the above figure, the input stream reads from the source and puts in the buffer (actually in programming, in a temporary variable, shown later). The output stream takes from the memory and writes to the destination.

All the classes needed to do with reading and writing (like file copying) are placed in the package `java.io` by the designers. All the I/O streams do the file reading or writing sequentially (means, one byte after another from start to the end of file). It can be done at random also. For this another class exists – `RandomAccessFile`. The `java.io` package also includes another class, `File`, to know the properties of a file like the file has read permission or write permission etc.



How Files and Streams Work:

Java uses **streams** to handle I/O operations through which the data is flowed from one location to another. For example, an **InputStream** can flow the data from a disk file to the internal memory and an **OutputStream** can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file. When we work with a text file, we use a **character** stream where one character is treated as per byte on disk. When we work with a binary file, we use a **binary** stream.



Stream

A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.

Three streams are created for us automatically:

- 1) System.out: standard output stream
- 2) System.in: standard input stream
- 3) System.err: standard error

Java encapsulates streams under `java.io` package, java defines two types of streams they are.

- 1) Byte stream.
- 2) Character stream.

1) Byte stream:

Byte stream represent data in the form of individual bytes, if a class name ends with a word 'stream', then it comes under byte stream. InputStream reads bytes and OutputStream writes bytes.

Exp:-

FileInputStream, FileOutputStream, BufferedInputStream, BufferedOutputStream.

Byte stream are used to handle any character (text), image, audio and video files for exp to store image files(.jpeg, gif) we should go for byte stream.

2) Character Stream:

It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

In the class name end with a word 'Reader' and 'Writer' then it taken as a text stream. Reader reads the text and Writer write the text.

Exp:-

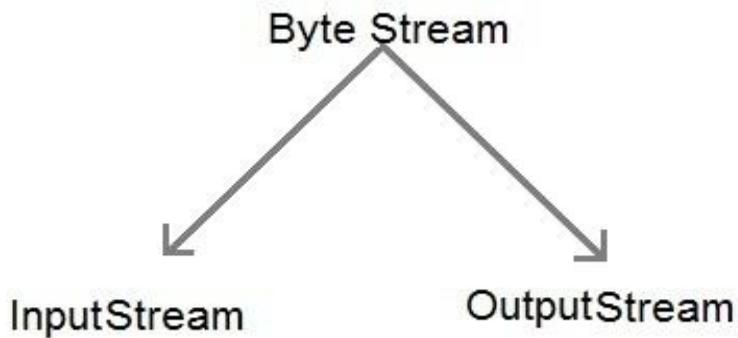
FileReader, FileWriter, BufferedReader, BufferedWriter.

1) Byte stream classes:

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.

To read and write 8-bit bytes, programs should use the byte streams, descendants of InputStream and OutputStream. InputStream and OutputStream provide the API and partial implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds. Two of the byte stream classes, ObjectInputStream and ObjectOutputStream, are used for object serialization. These classes are covered in Object Serialization.

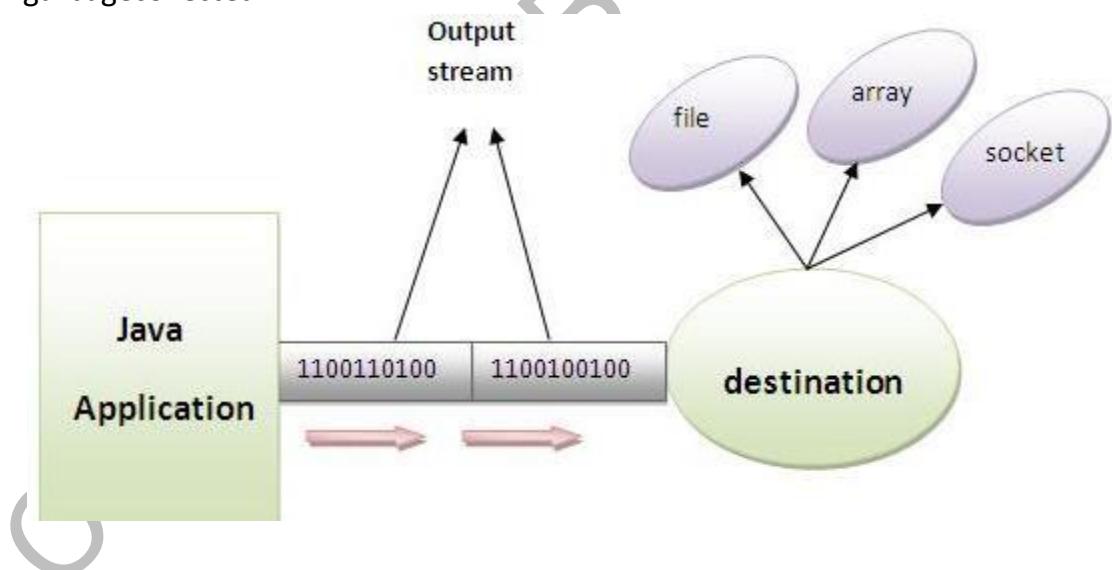
As with Reader and Writer, subclasses of InputStream and OutputStream provide specialized I/O that falls into two categories, as shown in the following class hierarchy figure: data sink streams (shaded) and processing streams (unshaded).

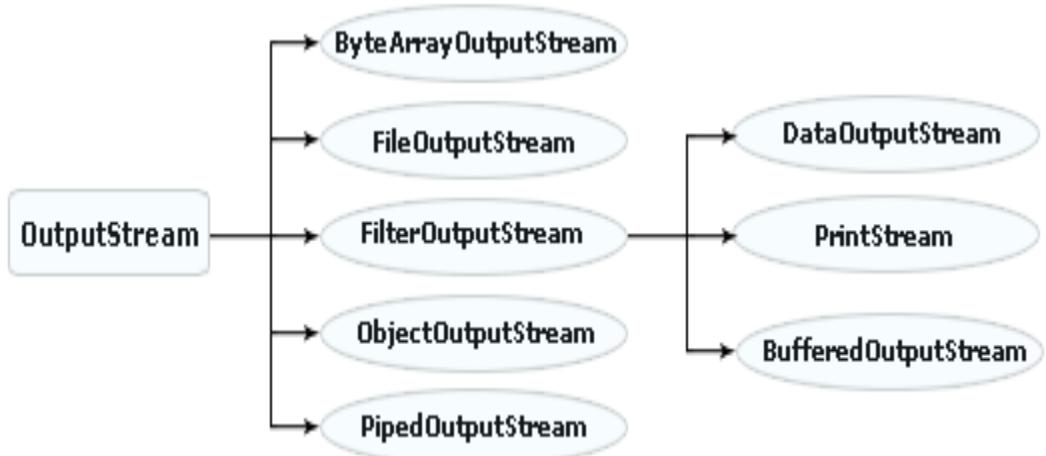


These two abstract classes have several concrete classes that handle various devices such as disk files, network connection

OutputStream:

The OutputStream class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the **close()** method, or let it be closed implicitly when the object is garbage collected.





OutputStream class

OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

methods of OutputStream class

1) public void write(int) throws IOException:

is used to write a byte to the current output stream.

2) public void write(byte[]) throws IOException:

is used to write an array of byte to the current output stream.

3) public void flush() throws IOException:

flushes the current output stream.

4) public void close() throws IOException:

is used to close the current output stream.

FileInputStream and FileOutputStream (File Handling):

FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

FileOutputStream class:

FileOutputStream is a byte-oriented stream used for writing stream of individual bytes into a particular file. To create file output stream object we should use the following constructors of file stream class.

If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

Constructors:

FileOutputStream(String filename):

Syn:- `FileOutputStream fos=new FileOutputStream("D:/nag.txt");`

Above syntax if the specified file not available under the specified location then JVM can take responsibility to create the file under the specified location.

FileOutputStream(String name, boolean b):

Syn:-

`FileOutputStream fos=new FileOutputStream("D:/nag.txt", true (or) false);`

Above if the b value is true old value is preserved in the file, if the b value is false old value is not preserved in the file.

FileOutputStream(File f):

Syn:-

`File f=new File("D:/nag.txt");`

`FileOutputStream fos=new FileOutputStream(f);`

In the above the specified file is available then only f is pointing that file. If the specified file is not there the JVM will create the new file.

FileOutputStream(File f, boolean b):

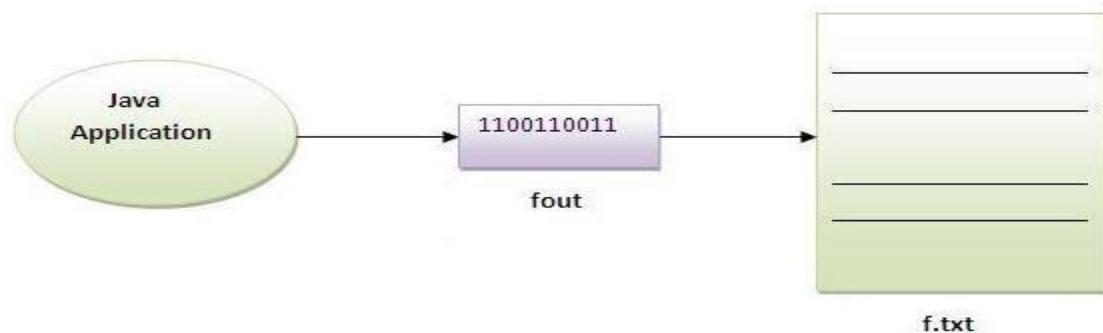
Exp:-

```
import java.io.*;
```

```

class FileDemo1
{
    public static void main(String[] args) throws IOException
    {
        FileOutputStream fos=new FileOutputStream("E://nag.txt");
        //FileOutputStream fout=new FileOutputStream("D:/n2.txt");
        fos.write(100);
        fos.write(105);
        String s1="welcome to nag";
        byte[] b1=s1.getBytes();
        fos.write(b1);
        System.out.println("data is inserted into files");
    }
}

```



Note:

- 1) Generally FileOutputStream is used for write streams of raw bytes that is image data.
- 2) If we are using FileOutputStream constructor must be handle FileNotFoundException using try and catch blocks (or) throws keyword.
- 3) If we using write() method must be handle IOException otherwise we will get Compile Time Error.

Exp:-

```
import java.io.*;
class FileOutDemo1
{
    public static void main(String[] args)throws IOException
    {
        FileOutputStream fos=new FileOutputStream("E:/nag4.txt",true);
        fos.write(100);
        fos.write(108);
        fos.write(220);
        byte[] b={10,20,30};
        fos.write(b);
        String s1="hi nag";
        String s2="hi arjun";
        byte[] b1=s1.getBytes();
        byte[] b2=s2.getBytes();
        fos.write(b1);
        fos.write(b2);
        System.out.println("data inserted");
    }
}
```

Exp:- Write the program to read the data from the keyboard and write any file

Exp:-

```
import java.io.*;

class CreateFileDataInput

{

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

    {

        DataInputStream dis=new DataInputStream(System.in);

        FileOutputStream fos=new FileOutputStream("E:/nag5.txt");

        System.out.println("eneter the text (9 at the end)");

        char ch;

        while((ch=(char)dis.read())!='9')

            fos.write(ch);

        fos.close();

    }

}
```

Output: 1234556666

665455669(whenever we enter 9 it will end and 9 will not store in the file)

Above program, we read data from the keyboard and write it to nag5.txt file. This program accept data from the keyboard till the user types '9' when he does not want to continue.

Reading the data from current file and write it into any another file

We can read the data of any file using the FileInputStream class whether it is java file, image file, video file etc. In this example, we are reading the data of nag6.txt file and writing it into another file n.txt file.

Exp:-

```
import java.io.*;
class ReadAndWrite
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis=new FileInputStream("E:/nag5.txt");
        FileOutputStream fos=new FileOutputStream("E:/n.txt");
        int i=0;
        while((i=fis.read())!=-1)
        {
            fos.write((byte)i);
        }
        fis.close();
        System.out.println("File read and write successfully");
    }
}
```

ObjectOutputStream

This class is having one method called `writeObject(Object o)` that takes object of any class, converts object into stream of bytes and writes it into a file or socket

ObjectInputStream and ObjectOutputStream

`ObjectInput` and `OutputStream` classes are used to serialize and deserialize class objects.

Object serialization

Object serialization is a process of converting object state (value of class instance variables is called as object state) into stream/series of bytes. The stream of bytes can be written to hard disk/into network.

With this feature in java programming language we can pass class instance state from one machine to another machine. Network programming is possible because of object serialization support in Java.

To convert object state into stream of bytes, create instance of a class, pass that object reference as argument to `ObjectOutputStream` class `writeObject()` method, `writeObject()` method converts state into stream of bytes and sends bytes to destination (file/network).

But before conversion `writeObject()` method verifies whether class is a sub class of `Serializable` interface or not.

If class is not a sub class of `Serializable` interface, then `writeObject()` method throws `NotSerializableException`. If sub class from `Serializable` interface then serialization takes place.

Exp:-

```
import java.io.*;

public class SerializeStudent implements Serializable
{
    int id;
    String name;

    public SerializeStudent(int id, String name)
    {
        this.id = id;
        this.name = name;
    }

    public static void main(String args[])throws Exception
    {
        SerializeStudent s1 =new SerializeStudent(101,"nag");
        FileOutputStream fout=new FileOutputStream("D:/n.txt");
        ObjectOutputStream out=new ObjectOutputStream(fout);
        out.writeObject(s1);
```

```
        out.flush();
        System.out.println("success");
    }
}
```

Output: Serialize the object Successfully

SocketOutputStream

write() method writes each char ascii value into socket as it is, on the other system SocketInputStream class read() method reads ascii value and return to client, client must type cast ascii value to original char.

ByteArrayOutputStream

ByteArrayOutputStream is the subclass of OutputStream which is created for writing the data into a byte array. This class has internal buffer which increases automatically as the data written into it. toByteArray() method is used to create a new byte array, and the toString() method is used to get the buffer's content as string. The close() method does not put any effect on the ByteArrayOutputStream, methods can be called after closing of the ByteArrayOutputStream.

Constructors

ByteArrayOutputStream():

This constructor is used to create a new byte array output stream with the initial buffer size of 32 bytes, size of this byte array output stream can be increases if the size of data written into it is increases.

public ByteArrayOutputStream():

ByteArrayOutputStream(int size) This constructor is used to create a new byte array output stream with the specified length of buffer size in bytes.

```
public ByteArrayOutputStream(int size)
```

Methods

close(): This method is used to close the ByteArrayOutputStream.

Syntax : public void close() throws IOException

reset(): This method is used to reset the counter i.e the count field of current ByteArrayOutputStream to zero.

Syntax : public void reset()

size(): This method is used to get the current size of buffer.

Syntax : public int size()

toByteArray(): This method is used to create a new byte array.

Syntax : public byte[] toByteArray()

toString(): This method is used to get the buffer's content as String.

Syntax : public String toString()

write(byte[] b, int off, int len):

This method is used to write the specified length of byte from the specified byte array started from the specified offset.

Syntax : public void write(byte[] b, int off, int len)

write(int b): This method is used to write a single byte to the current byte array output stream.

Syntax : public void write(int b)

writeTo(OutputStream out):

This method is used to write all contents of byte array to the output stream given as argument.

Syntax : public void writeTo(OutputStream out) throws IOException

Exp:-

```
/* This example demonstrate that how a byte  
stream can be written to the byte array  
output stream one at a time*/
```

```
import java.io.ByteArrayOutputStream;  
  
import java.io.IOException;  
  
public class WriteByteStream
```

```
{  
public static void main(String args[])  
{  
    ByteArrayOutputStream baos = null;  
    int bufsize;  
    String bufferStr;  
    try  
    {  
        baos = new ByteArrayOutputStream();  
        byte[] b;  
        String str = args[0];  
        b = str.getBytes();  
        baos.write(b);  
        System.out.println("Input String is : " + baos.toString());  
        System.out.println("Number of characters in "+ str +" is : " + baos.size());  
        byte[] bufArray = baos.toByteArray();  
        System.out.println("Converted String to a byte array : ");  
        for(int i=0; i<bufArray.length; i++)  
        {  
            System.out.print((char)bufArray[i]);  
            System.out.print(",");  
        }  
        System.out.println();  
        baos.reset();
```

```
bufsize = baos.size();

System.out.println("Size after reset: " + bufsize);

}

catch(IOException e)

{

System.out.println("IOException caught..!!");

e.printStackTrace();

}

}

}
```

Output: E:\COREJAVA\Files>java WriteByteStream hello

Input String is : hello

Number of characters in 'hello' is : 5

Converted String to a byte array :

h,e,l,l,o,

Size after reset: 0

But the main drawback of these I/O classes are:

because of writing data byte-by-byte number of write() operations increases, and application performance reduces.

In OutputStream also one **FilterOutputStream** class is provided. subclasses of this class makes write operations faster.

BufferedOutputStream

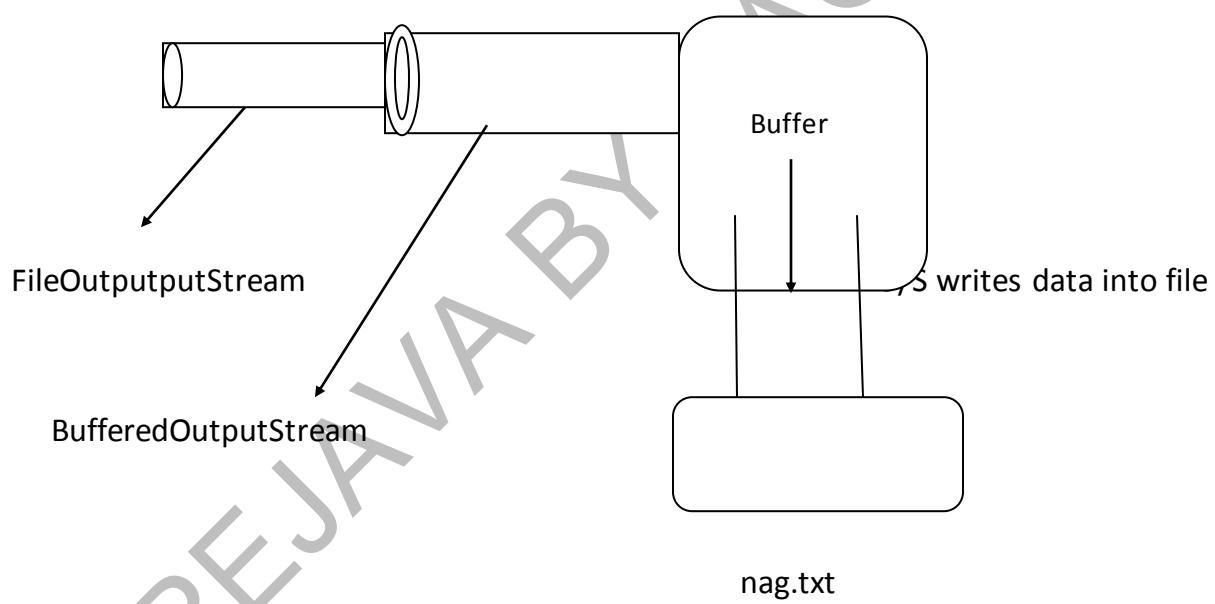
Improving Efficiency using BufferedOutputStream

Normally , whenever we write data into a file using FileInputStream as

```
FileOutputStream fos=new FileOutputStream("E:/nag5.txt");  
fos. write(ch);
```

We call write() method to write a character (ch) into the file. The Operating System is invoked to write the character into the file. So, if we write 10 characters, the underlying Operating System would be called 10 times and it will write the character into the file. This waste a lot of time.

On the other hand, if Buffered classes are used, they provide a buffer(temporary block of memory), which is first filled with characters and then characters from the buffer are at once written into the file by the Operating System. This Takes less time and hence efficiency is more.



Here , the buffer size is declared as 1024 bytes. If the buffer size is not specified, then a default buffer size of 512 bytes is used.

Exp:-

```
import java.io.*;  
  
class BufferedCreateFile  
{
```

```

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

{
    DataInputStream dis = new DataInputStream(System.in);

    FileOutputStream fos = new FileOutputStream("E:/nag6.txt", true);

    BufferedOutputStream bos = new BufferedOutputStream(fos, 1024);

    System.out.println("enter the text (@ at the end)");

    char ch;

    while ((ch = (char) dis.read()) != '@')

        bos.write(ch);

    bos.close();

}
}

```

DataOutputStream

DataOutput : An interface that facilitate to convert the primitive types values of Java to a series object bytes write byte to the binary stream.

There are several classes which implements the DataOutput interface. Some of them are as follows :

Classes	Description
DataOutputStream	DataOutputStream allows your application to write the Java primitive data type values to the output stream.
RandomAccessFile	This class provides the facility to read and write to a random access file.
ObjectOutputStream	This class is used to write the primitive data types and graphs of Java objects into an output stream.

Methods of DataOutput

- ❖ **write(byte[] b)** : This method is used to write all the bytes of a specified array (say, b is an array) into the output stream.

```
void write(byte[] b) throws IOException
```

- ❖ **write(byte[] b, int off, int len)** : This method is used to write the specified number of bytes from the specified array into the output stream.▶

```
void write(byte[] b, int off, int len) throws IOException
```

- ❖ **write(int b)** : This method is used to write for the integer argument to the output stream.

```
void write(int b) throws IOException
```

- ❖ **writeBoolean(boolean v)** :

```
void writeBoolean(boolean v) throws IOException
```

- ❖ **writeByte(int v)** : This method is used to write for the integer argument to the output stream but the written byte using this method can be read by readByte() method of DataInput.

```
void writeByte(int v) throws IOException
```

- ❖ **writeBytes(String s)** : This method is used to write string, in order to one byte, into the output stream.

```
void writeBytes(String s) throws IOException
```

- ❖ **writeChar(int v)** : This method is used to write the char value that made up from the two bytes, into the output stream.

```
void writeChar(int v) throws IOException
```

- ❖ **writeChars(String s)** : This method is used to write each characters of a specified string in order to two bytes per character, into the output stream.

```
void writeChars(String s) throws IOException
```

- ❖ **writeDouble(double v)** : This method is used to write the double value that made up from the eight bytes into the output stream.

```
void writeDouble(double v) throws IOException
```

- ❖ **writeFloat(float v)** : This method is used to write the float value that made up from the four bytes into the output stream.▶

```
void writeFloat(float v) throws IOException
```

- ❖ **writeInt(int v)** : This method is used to write the int value that made up from the four bytes into the output stream.

```
void writeInt(int v) throws IOException
```

- ❖ **writeLong(long v)** : This method is used to write the long value that made up from the eight bytes into the output stream.

```
void writeLong(long v) throws IOException
```

- ❖ **writeShort(int v)** : This method is used to write the two bytes into the output stream for displaying the argument's value

```
void writeShort(int v) throws IOException
```

- ❖ **writeUTF(String s)** : This method is used to write the string s into the output stream.

```
void writeUTF(String s) throws IOException
```

Exp:-

```
import java.io.*;  
  
class DataOutPutStream  
{  
  
    public static void main(String[] args)
```

```
{  
    try  
    {  
        FileOutputStream outFile = new FileOutputStream("E:/M1.txt");  
        DataOutputStream outStream = new DataOutputStream(outFile);  
        outStream.writeBoolean(true);  
        outStream.writeInt(123456);  
        outStream.writeChar('A');  
        outStream.writeDouble(9999.99);  
        System.out.println(outStream.size ()+" bytes were written");  
        outStream.close ();  
        outFile.close ();  
    }  
    catch (Exception e)  
    {  
        System.out.println(e);  
    }  
}
```

PrintStream

The PrintStream class is obtained from the FilterOutputStream class that implements a number of methods for displaying textual representations of Java primitive data types. It adds the functionality to another output streams that has the ability to print various data values conveniently. Unlike other output streams, a PrintStream never throws an

IOException and the data is flushed to a file automatically i.e. the flush method is automatically invoked after a byte array is written,

The PrintStream is designed to print data at DOS prompt with its print() and println() methods. This stream comes with the ability of automatic flushing of data. In contrary to the methods of all I/O streams, the methods of PrintStream class do not PrintStream methods are deprecated in favor of PrintWriter, but not System.out. There exists a similar class on character streams side, PrintWriter with println() method.

The println() and print() methods are overloaded many times in the PrintStream class to accept different data types and objects as parameters. println() method is overloaded 10 times and print() method is overloaded 9 times.

constructor

```
PrintStream (java.io.OutputStream out); //create a new print stream
```

The print() and Println() methods of this class give the same functionality as the method of standard output stream and follow the representations with newline.

Exp:-

```
import java.io.*;  
  
class PrintStreamDemo {  
  
    public static void main(String args[]){  
  
        FileOutputStream out;  
  
        PrintStream ps; // declare a print stream object  
  
        try {  
  
            // Create a new file output stream  
  
            out = new FileOutputStream("E:/myfile.txt");  
  
            // Connect print stream to the output stream  
  
            ps = new PrintStream(out);  
  
            ps.println ("This data is written to a file:");
```

```
        System.err.println ("Write successfully");

        ps.close();

    }

    catch (Exception e){

        System.out.println ("Error in writing to file");

    }

}

}
```

Output:

This data is written to a file

Note: This program firstly create an object "ps" of the PrintStream class the specified data is written through that object using the println() method.

Exp:-

```
import java.io.*;

public class PSDemo

{

    public static void main(String args[]) throws IOException

    {

        PrintStream pstream = new PrintStream(System.out);

        pstream.println(100);

        pstream.println("World");

        pstream.print(false);

        pstream.println(new Double(10.5));

        System.out.println("Java is simple but ocean");

        System.err.println("Practice it carefully");
```

```
    }  
}  
}
```

Output:-

100

World

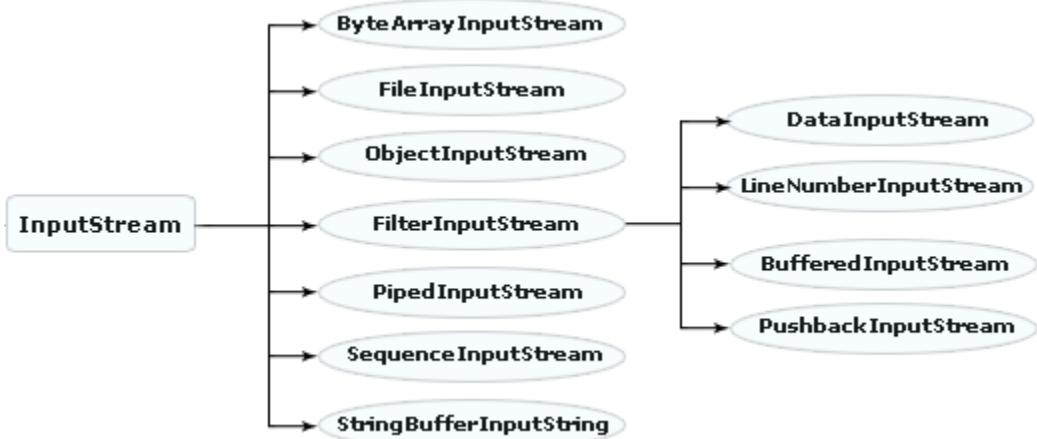
false10.5

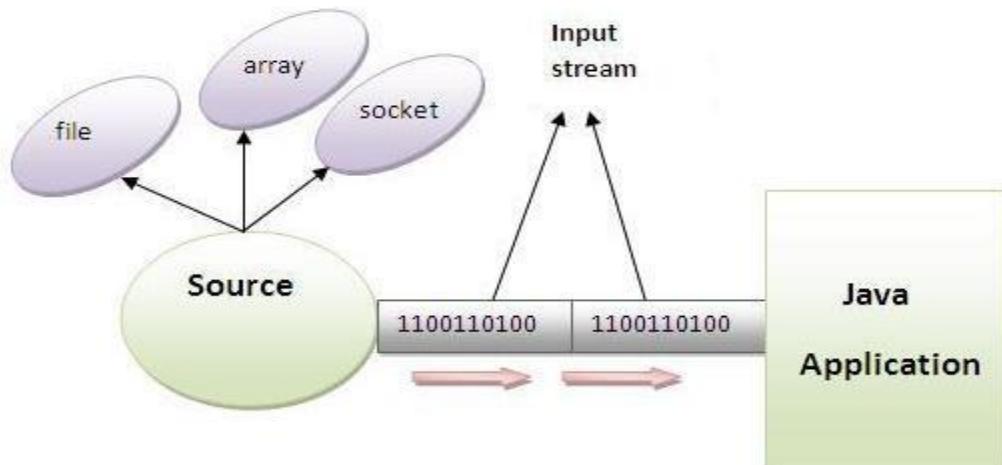
Java is simple but ocean

Practice it carefully

InputStream:

The **InputStream** class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a **file**, a **string**, or **memory** that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when you create it. You can explicitly close a stream with the **close()** method, or let it be closed implicitly when the object is found as a garbage.





InputStream class

InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

methods of InputStream class

1) public abstract int read()throws IOException:

reads the next byte of data from the input stream. It returns -1 at the end of file.

2) public int available()throws IOException:

returns an estimate of the number of bytes that can be read from the current input stream.

3) public void close()throws IOException:

is used to close the current input stream.

FileInputStream

It is byte oriented stream used to read the data from a file to our java program. To create the FileInputStream object we should use the following constructors from File input class.

Constructors:

1) FileInputStream(String Filename)

2) FileInputStream(File f)

Note: If the specified file is not available then JVM raise an exception
java.lang. FileNotFoundException.

Syn:-

```
FileInputStream fis=new FileInputStream("E:/nag4.txt");
```

Exp:-

```
import java.io.*;  
  
class FileInDemo1  
{  
  
    public static void main(String[] args)throws IOException {  
  
        FileInputStream fis=new FileInputStream("E:/nag6.txt");  
  
        int i=fis.read();  
  
        while(i!=-1)  
        {  
  
            System.out.println((char)i+"---->" +i);  
  
            i=fis.read();  
        }  
        System.out.println(i);  
    }  
}
```

Output:

n---->110

a---->97

g---->103

a---->97

r---->114

j---->106

u---->117

n---->110

a---->97

g---->103

a---->97

d---->100

d---->100

a---->97

m---->109

SequenceInputStream:

class SequenceInputStream is used to copy a number of source files to one destination file. Many input streams are concatenated and converted into a single stream and copied at a stretch to the destination file.

reads the first file until reached at end the of file and then reads another file until reached at the end of file and so on.

Merging of multiple files can be done easily in Java using SequenceInputStream. SequenceInputStream is a good example to show that Java is a production language where with less code a greater extent of functionality is realized.

Constructors of SequenceInputStream

SequenceInputStream(Enumeration<? extends InputStream> e)

SequenceInputStream(InputStream s1, InputStream s2)

Commonly used methods of SequenceInputStream

available() :

Readable number of bytes are returned by this method.

close() :

This method is used to close the input stream and it releases the resources used by the stream.

read() :

This method is used to read the bytes from the input stream.

read(byte[] b, int off, int len) :

This method is used to read the specified length 'len' of bytes from the specified byte array 'b' started form the offset 'off'.

Just Merging of Two Files

In the following program, take any two files existing in the same directory and pass them to two different FileInputStream constructors. The contents of these two source files are displayed at DOS prompt and also copied to a destination file.

Exp:-

```
import java.io.*;
class SequenceInput
{
    public static void main(String args[])throws Exception
    {
        FileInputStream fin1=new FileInputStream("E:/nag5.txt");
        FileInputStream fin2=new FileInputStream("E:/nag6.txt");
        SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
        int i;
```

```
    while((i=sis.read())!=-1)

    {
        System.out.println((char)i);
    }
}
```

SequenceInputStream class that reads the data from two files and write it into another

Exp:-

```
import java.io.*;

class SequenceInput
{
    public static void main(String args[])throws Exception
    {
        FileInputStream fin1=new FileInputStream("E:/nag5.txt");
        FileInputStream fin2=new FileInputStream("E:/nag6.txt");
        SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
        FileOutputStream fout=new FileOutputStream("E:/M.java");
        int i;
        while((i=sis.read())!=-1)
        {
            fout.write(i);
            System.out.println((char)i);
        }
    }
}
```

```
        System.out.println("File created successfully");

        fout.close();

        sis.close();

        fin1.close();

        fin2.close();

    }

}
```

Output:

File Created successfully

SequenceInputStream class that reads the data from multiple files using enumeration

If we need to read the data from more than two files, we need to have these information in the Enumeration object. Enumeration object can be get by calling elements method of the Vector class. Let's see the simple example where we are reading the data from the 4 files.

Exp:-

```
import java.io.*;
import java.util.*;
class SequenceEnumeration
{
    public static void main(String args[])throws IOException
    {
        //creating the FileInputStream objects for all the files
        FileInputStream fin=new FileInputStream("E:/n.txt");
        FileInputStream fin2=new FileInputStream("E:/nag5.txt");
        FileInputStream fin3=new FileInputStream("E:/nag6.txt");
```

```
FileInputStream fin4=new FileInputStream("E:/nag4.txt");

//creating Vector object to all the stream

Vector v=new Vector();

v.add(fin);

v.add(fin2);

v.add(fin3);

v.add(fin4);

//creating enumeration object by calling the elements method

Enumeration e=v.elements();

//passing the enumeration object in the constructor

SequenceInputStream bin=new SequenceInputStream(e);

int i=0;

while((i=bin.read())!=-1)

{

System.out.print((char)i);

}

bin.close();

fin.close();

fin2.close();

}

}
```

ByteArrayInputStream

The `java.io.ByteArrayInputStream` class contains an internal buffer that contains bytes that may be read from the stream. This class is very useful to read data from byte arrays.

In UDP protocol, the data is received in byte array and `ByteArrayInputStream` is helpful to read data from this array.

In this program, the data is read fully from the byte array. Each byte is read and printed along with its ASCII value.

Exp:-

```
import java.io.*;

public class ByteArrayInputStreamExp

{
    public static void main(String args[]) throws IOException
    {
        byte vowels[] = { 'a', 'e', 'i', 'o', 'u' };

        ByteArrayInputStream bai = new ByteArrayInputStream(vowels);

        int temp;

        while( ( temp = bai.read() ) != -1 )
        {
            char ch = (char) temp;

            System.out.println("Vowel: " + ch + " and its ASCII value: " + temp);
        }

        System.out.println("\n\nTo read again after calling reset()
method");

        bai.reset();

        // places the file pointer at the beginning of the array
```

```
        while( ( temp = bai.read() ) != -1)

        {

            char ch = (char) temp; System.out.println("Vowel: " + ch + " and its
            ASCII value: " + temp);

        }

        bai.close();

    }

}
```

Output:

Vowel: a and its ASCII value: 97

Vowel: e and its ASCII value: 101

Vowel: i and its ASCII value: 105

Vowel: o and its ASCII value: 111

Vowel: u and its ASCII value: 117

To read again after calling reset() method

Vowel: a and its ASCII value: 97

Vowel: e and its ASCII value: 101

Vowel: i and its ASCII value: 105

Vowel: o and its ASCII value: 111

Vowel: u and its ASCII value: 117

ByteArrayInputStream bai = new ByteArrayInputStream(vowels);

The byte array vowels is passed as parameter to the ByteArrayInputStream constructor.

temp = bai.read()

char ch = (char) temp;

The `read()` method of `ByteArrayInputStream` (inherited from its super class, `InputStream`) reads byte by byte from the array, converts the each byte into its ASCII integer value and returns. For example, the `read()` method reads 'a' and returns it as 97. To print the original character, the temp variable is explicitly type casted to char ch.

```
bai.reset();
```

The `reset()` method places the file pointer at the beginning of the array, so that, if necessary we can read the data again. The same thing is done in the program; the data is read again and printed (same output).

```
byte vowels[] = { 'a', 'e', 'i', 'o', 'u' };
```

The above statement can be replaced with the following (where ASCII values are taken instead of characters) and can be obtained the same output.

SocketInputStream

`SocketInputStream` class is having `read()` method to read stream of bytes from socket (n/w programming)

StringBufferInputStream

As in the earlier program, where the source of reading is a byte array in case now the source is a string buffer in case of `StringBufferInputStream`. That is, `StringBufferInputStream` reads from a string buffer (or the other way, string).

This class methods and constructors are Deprecated (observe the compiler message in screen shot) in favor of `StringReader` of character streams as designers observed problems in converting characters to bytes.

Exp:-

```
import java.io.*;  
  
public class SBDemo  
{  
  
    public static void main(String args[]) throws IOException  
    {  
  
        String comments = "Improve\\nthe\\ncontent\\nwith\\nyour\\nvalueble\\ncomments";
```

```
StringBufferInputStream sbis = new StringBufferInputStream(comments);

DataInputStream dis = new DataInputStream(sbis);

String str; while( ( str = dis.readLine() ) != null )

{

    System.out.println(str);

}

dis.close();

sbis.close();

}

}
```

ObjectInputStream

ObjectInputStream class is having readObject() that reads objects from file/network.

ObjectInputStream and ObjectOutputStream

ObjectInput and OutputStream classes are used to serialize and deserialize class objects.

Object serialization

Object serialization is a process of converting object state (value of class instance variables is called as object state) into stream/series of bytes. The stream of bytes can be written to hard disk/into network.

With this feature in java programming language we can pass class instance state from one machine to another machine. Network programming is possible because of object serialization support in Java.

To convert object state into stream of bytes, create instance of a class, pass that object reference as argument to ObjectOutputStream class writeObject() method, writeObject() method converts state into stream of bytes and sends bytes to destination (file/network).

But before conversion writeObject() method verifies whether class is a sub class of Serializable interface or not.

If class is not a sub class of Serializable interface, then writeObject() method throws NotSerializableException. If sub class from Serializable interface then serialization takes place.

Exp:-

```
import java.io.*;  
  
class Depersist{  
  
    public static void main(String args[])throws Exception{  
  
        ObjectInputStream in=new ObjectInputStream(new FileInputStream("D:/n.txt"));  
  
        SerializeStudent s=(SerializeStudent)in.readObject();  
  
        System.out.println(s.id+" "+s.name);  
  
        in.close();  
  
    }  
}
```

Output: Deserialize the object successfully

PipedInputStream and PipedOutputStream classes

The PipedInputStream and PipedOutputStream classes can be used to read and write data simultaneously. Both streams are connected with each other using the connect() method of the PipedOutputStream class.

Exp:-

```
import java.io.*;  
  
public class PipedInputStreamDemo {  
  
    public static void main(String[] args) {  
  
        // create a new Piped input and Output Stream  
  
        PipedOutputStream out = new PipedOutputStream();
```

```
PipedInputStream in = new PipedInputStream();

try {
    // connect input and output
    in.connect(out);

    // write something
    out.write(70);
    out.write(71);

    // read what we wrote
    for (int i = 0; i < 2; i++) {
        System.out.println("'" + (char) in.read());
    }
}

catch (IOException ex) {
    ex.printStackTrace();
}
}
```

Output:

F

G

Exp:- Here, we have created two threads t1 and t2. The **t1** thread writes the data using the PipedOutputStream object and the **t2** thread reads the data from that pipe using the PipedInputStream object. Both the piped stream object are connected with each other.

```
import java.io.*;
```

```
class PipedWR{  
  
    public static void main(String args[])throws Exception{  
  
        final PipedOutputStream pout=new PipedOutputStream();  
  
        final PipedInputStream pin=new PipedInputStream();  
  
        pout.connect(pin);//connecting the streams  
  
        //creating one thread t1 which writes the data  
  
        Thread t1=new Thread(){  
  
            public void run(){  
  
                for(int i=85;i<=90;i++){  
  
                    try{  
  
                        pout.write(i);  
  
                        Thread.sleep(500);  
  
                    }  
  
                    catch(Exception e){}  
  
                }  
  
            }  
  
        };  
  
        //creating another thread t2 which reads the data  
  
        Thread t2=new Thread(){  
  
            public void run(){  
  
                try{  
  
                    for(int i=85;i<=90;i++)  
  
                        System.out.println(pin.read());  
  
                }catch(Exception e){}

```

```
    }

};

//starting both threads

t1.start();

t2.start();

}

}
```

Output:

```
85
86
87
88
89
90
```

BufferedInputStream

BufferedInputStream class is having read() method that is capable of reading 512 bytes of data in a single read() operation. We can increase or reduce the size of buffer by passing buffer size as argument in its constructor.

Exp:-

```
import java.io.*;
class BufferInput
{
public static void main(String[] args)
{
try
{
FileInputStream fin=new FileInputStream("E:/nag6.txt");
BufferedInputStream bin=new BufferedInputStream(fin);
int i;
while((i=bin.read())!=-1)
```

```
        {
            System.out.print((char)i);
        }
        bin.close();
        fin.close();
    }
}
catch (Exception e)
{
    System.out.println(e);
}
}
```

DataInputStream:

Exp:-

```
import java.io.*;
class DataInputStreamExp
{
    public static void main(String[] args)
    {
        try
        {

            FileInputStream inFile = new FileInputStream("E:/M1.txt");
            DataInputStream inStream = new DataInputStream(inFile);
            System.out.println(inStream.readBoolean ());
            System.out.println(inStream.readInt ());
            System.out.println(inStream.readChar ());
            System.out.println(inStream.readDouble ());
            inStream.close ();
            inFile.close ();
        }
        catch (IOException e)
        {
```

```
        System.out.println(e);
    }
}
}
```

Output:

```
true  
123456  
A  
9999.99
```

PushbackInputStream

The PushbackInputStream class provides a pushback buffer so a program can “unread” bytes. In other words, it can add bytes to the stream and then read them. This class allows to add data to the stream while they’re reading it. The next time data is read from the stream, the unread bytes are reread.

Exp:-

```
import java.io.IOException;  
  
import java.io.PushbackInputStream;  
  
public class PushBackInput {  
  
    public static void main(String[] args) throws IOException {  
  
        PushbackInputStream in = new PushbackInputStream(System.in, 3);  
  
        in.unread(0);  
  
        in.unread(5);  
  
        in.unread(10);  
  
        System.out.println(in.read());  
  
        System.out.println(in.read());  
  
        System.out.println(in.read());  
    }  
}
```

```
}
```

Output:

```
10
```

```
5
```

```
0
```

LineNumberInputStream

It is a subclass of FileInputStream stream. We know each filter stream adds some extra functionality. The LineNumberInputStream does the extra work of adding line numbers.

Exp:-

```
import java.io.*;  
  
public class LISDemo  
{  
    public static void main(String args[]) throws IOException  
    { // reading-side  
  
        String str1 =  
        "The\\nonly\\ntrue\\nwisdom\\nis\\nin\\nknowing\\nyou\\nknow\\nnothing\\n";StringBufferIn  
putStream sbstream = new StringBufferInputStream(str1); // low-level  
  
        LineNumberInputStream listream = new LineNumberInputStream(sbstream); // high-  
level  
  
        DataInputStream distream = new DataInputStream(listream); // high-level  
                // writing-side  
  
        FileOutputStream fostream = new FileOutputStream("E:/Quotes.txt"); // low-level  
  
        BufferedOutputStream bostream = new BufferedOutputStream(fostream); // high-level  
  
        String str;  
  
        while( ( str = distream.readLine() ) != null)
```

```
{  
    String s1 = listream.getLineNumber() + ". " + str;  
    System.out.println(s1);  
}  
  
bostream.close(); fostream.close();  
  
distream.close(); listream.close(); sbstream.close();  
}  
}
```

Output:

1. The
2. only
3. true
4. wisdom
5. is
6. in
7. knowing
8. you
9. know
10. nothing

Note: getLineNumber() method of LineNumberInputStream returns numbers that increments automatically. readLine() method of DataInputStream reads each line from string str1, delimited by \n, and getLineNumber() adds the line number.

When the above program is compiled, the compiler raises a warning message (see the above screen shot) as **readLine() method of DataInputStream is deprecated.**

A similar class on character streams is **LineNumberReader** which gives same functionality of adding line numbers.

LineNumberReader

The character stream LineNumberReader is equivalent to LineNumberInputStream on byte streams side. Being a subclass of BufferedReader, it can make use of all the methods of BufferedReader like readLine() to read a line. getLineNumber() method of this class can be used to give line numbers.

Following is the class signature:

```
public class LineNumberReader extends BufferedReader
```

Exp:-

```
import java.io.*;

public class LNRDemo

{
    public static void main(String args[]) throws IOException
    {
        FileReader freader = new FileReader("E:/nag.txt");

        LineNumberReader Inreader = new LineNumberReader(freader);

        String s1; while( ( s1 = Inreader.readLine() ) != null)
        {
            System.out.println(Inreader.getLineNumber() + ":" + s1);
        }

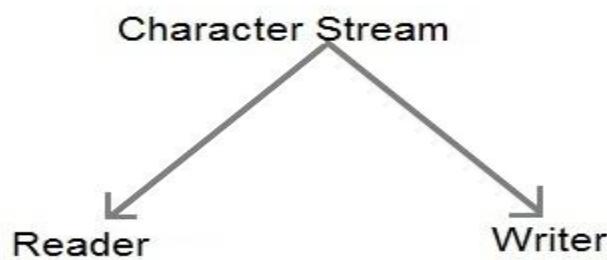
        Inreader.close(); freader.close();
    }
}
```

Output:

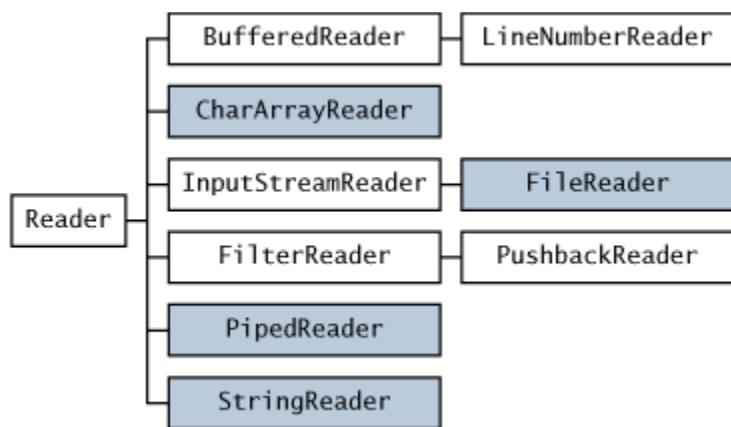
1. Welcome
2. to
3. nag

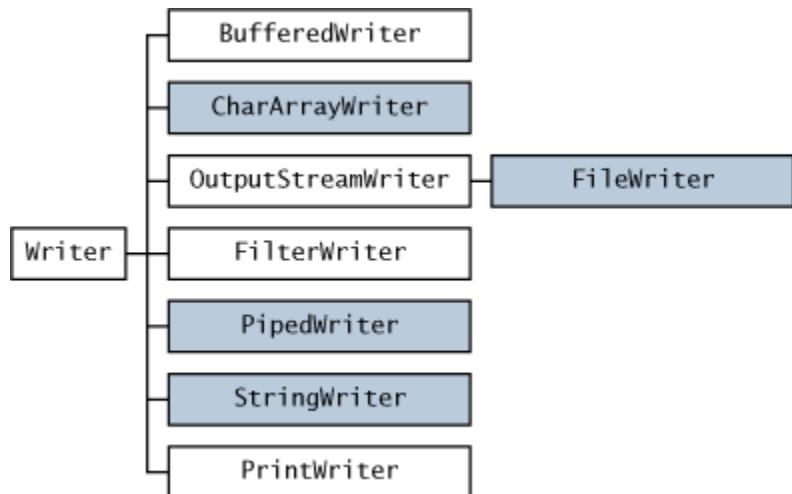
Character Stream

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



These two abstract classes have several concrete classes that handle Unicode character. Reader and Writer are the abstract super classes for character streams in `java.io`. Reader provides the API and partial implementation for readers — **streams that read 16-bit characters** — and Writer provides the API and partial implementation for writers — **streams that write 16-bit characters**. Subclasses of Reader and Writer implement specialized streams and are divided into two categories: those that read from or write to data sinks (shown in gray in the following figures) and those that perform some sort of processing (shown in white).





FileWriter:

FileWriter class is used to write character-oriented data to the file. Sun Microsystems has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

This class inherits from the OutputStreamWriter class. The class is used for writing streams of characters.

FileWriter: FileWriter is the simplest way to write a file in java, it provides overloaded write method to write int, byte array and String to the File. You can also write part of the String or byte array using FileWriter. FileWriter writes directly into Files and should be used only when number of writes are less.

BufferedWriter: BufferedWriter is almost similar to FileWriter but it uses internal buffer to write data into File. So if the number of write operations are more, the actual IO operations are less and performance is better. You should use BufferedWriter when number of write operations are more.

FileOutputStream: FileWriter and BufferedWriter are meant to write text to the file but when you need raw stream data to be written into file, you should use FileOutputStream to write file in java.

Files: Java 7 introduced Files utility class and we can write a file using it's write function, internally it's using OutputStream to write byte array into file.

Exp:-

```
import java.io.*;

class FileWrite

{

    public static void main(String args[])

    {

        try{

            // Create file

            FileWriter fstream = new FileWriter("E:/out.txt");

            BufferedWriter out = new BufferedWriter(fstream);

            out.write("Hello Java");

            System.out.println("File created successfully");

            //Close the output stream

            out.close();

        }catch (Exception e){//Catch exception if any

            System.err.println("Error: " + e.getMessage());

        }

    }

}
```

FileReader:

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.

Java FileReader is character streams used to read files in the same way as in FileOutputStream. Instead of bytes we use characters.

Constructor

FileReader(File file): This constructor creates a FileReader which contains a reference of File to read the stream.

Syntax : public FileReader(File file) throws FileNotFoundException

FileReader(FileDescriptor fd): This constructor creates a FileReader which contains a reference of FileDescriptor to read the stream.

Syntax : public FileReader(FileDescriptor fd)

FileReader(String fileName): This constructor creates a FileReader which contains the name of a file to read the stream.

Syntax : public FileReader(String fileName) throws FileNotFoundException

Exp:-

```
import java.io.*;  
  
class FileReaderExp  
{  
    public static void main(String args[])throws Exception  
    {  
        FileReader fr=new FileReader("E:/out.txt");  
        int i;  
        System.out.println();  
        System.out.println("***** OUTPUT *****");  
        System.out.println();  
        while((i=fr.read())!=-1)  
        System.out.print((char)i);
```

```
        System.out.println();
        fr.close();
    }
}
```

Exp:-

```
import java.io.Reader;
import java.io.FileReader;
import java.io.IOException;
public class JavaFileReaderExample
{
    public static void main(String args[])
    {
        FileReader fr = null;
        try
        {
            fr = new FileReader("E:/out.txt");
            int c ;
            System.out.println();
            System.out.println("***** OUTPUT *****");
            System.out.println();
            while((c= fr.read()) != -1)
            {
                System.out.print((char) c);
            }
        }
```

```
        System.out.println();  
    }  
  
    catch(Exception ex)  
    {  
        System.out.println(ex);  
    }  
  
    finally  
    {  
        if(fr != null)  
        {  
            try  
            {  
                fr.close();  
            }  
            catch(IOException ioe)  
            {  
                System.out.println(ioe);  
            }  
        }  
    }  
}// end finally  
}// end main  
}// end class
```

Reading the data from the Keyboard:

There are many ways to read data from the keyboard. For example:

InputStreamReader

Console

Scanner

DataInputStream etc.

InputStreamReader class:

InputStreamReader class can be used to read data from keyboard. It performs two tasks:

connects to input stream of keyboard

converts the byte-oriented stream into character-oriented stream

BufferedReader class:

BufferedReader class can be used to read data line by line by readLine() method.

Example of reading data from keyboard by InputStreamReader and BufferedReader class:

In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

Exp:-

```
import java.io.*;  
  
class InputReaderFromKeyBoard  
{  
    public static void main(String args[])throws Exception  
  
        InputStreamReader r=new InputStreamReader(System.in);  
        BufferedReader br=new BufferedReader(r);  
        System.out.println("Enter ur name");  
        String name=br.readLine();
```

```
        System.out.println("Welcome "+name);

    }

}
```

Output:

Enter ur name

arjun

Welcome arjun

Console class:

The Console class can be used to get input from the keyboard.

How to get the object of Console class:

System class provides a static method named `console()` that returns the unique instance of Console class.

Syntax:

```
public static Console console(){}
```

Commonly used methods of Console class:

1) public String readLine():

is used to read a single line of text from the console.

2) public String readLine(String fmt, Object... args):

it provides a formatted prompt then reads the single line of text from the console.

3) public char[] readPassword():

is used to read password that is not being displayed on the console.

4) public char[] readPassword(String fmt, Object... args):

it provides a formatted prompt then reads the password that is not being displayed on the console.

Console class that reads name of user:

Exp:-

```
import java.io.*;

class ConsoleExp

{

public static void main(String args[])

{

    Console c=System.console();

    System.out.println("Enter ur name");

    String n=c.readLine();

    System.out.println("Welcome "+n);

}

}
```

Exp:-

```
import java.io.*;

class ConsoleExp1

{

public static void main(String args[])

{

    Console c=System.console();

    System.out.println("Enter password");

    char[] ch=c.readPassword();

    System.out.println("Password is");

    for(char ch2:ch)
```

```
        System.out.print(ch2);

    }

}
```

Scanner class:

One of the features of Java is "robust". It does not allow the programmer, to the maximum extent, to commit mistakes. For example, giving the size of the array in array initialization is a compilation error. Java designers included many methods in Scanner class intended to check the input whether it is a byte, short, long and double etc. For example, reading with `nextInt()` when user enters a double is an exception like `InputMismatchException`. So the programmer should take every care to see what the user entered and accordingly use appropriate `nextXXX()` method to read.

The checking methods are of type `hasNextXXX()` like `hasNext()`, `hasNextBoolean()`, `hasNextShort()`, `hasNextInt()` and `hasNextDouble()` etc. that checks the input is a word, a boolean, a short, an integer or a double.

The Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

Commonly used methods of Scanner class:

There is a list of commonly used Scanner class methods:

public String next():

it returns the next token from the scanner.

public String nextLine():

it moves the scanner position to the next line and returns the value as a string.

public byte nextByte():

it scans the next token as a byte.

public short nextShort():

it scans the next token as a short value.

public int nextInt():

it scans the next token as an int value.

public long nextLong():

it scans the next token as a long value.

public float nextFloat():

it scans the next token as a float value.

public double nextDouble():

it scans the next token as a double value.

Exp:-

```
import java.util.Scanner;  
  
class ScannerTest  
{  
  
    public static void main(String args[])  
    {  
  
        Scanner sc=new Scanner(System.in);  
  
        System.out.println("Enter your rollno");  
  
        int rollno=sc.nextInt();  
  
        System.out.println("Enter your name");  
  
        String name=sc.next();  
  
        System.out.println("Enter your fee");  
  
        double fee=sc.nextDouble();  
  
        System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);  
  
    }  
}
```

Output:

Enter your rollno

39

Enter your name

nag

Enter your fee

50000

Rollno:39 name:nag fee:50000.0

Compressing and Uncompressing File

The DeflaterOutputStream and InflaterInputStream classes provide mechanism to compress and uncompress the data in the deflate compression format.

DeflaterOutputStream class:

The DeflaterOutputStream class is used to compress the data in the deflate compression format. It provides facility to the other compression filters, such as GZIPOutputStream.

Compressing file using DeflaterOutputStream class

In this example, we are reading data of a file and compressing it into another file using DeflaterOutputStream class. You can compress any file, here we are compressing the nag5.txt file.

Exp:-

```
import java.io.*;
import java.util.zip.*;
class Compress
{
    public static void main(String args[])
    {
```

```
try
{
    FileInputStream fin=new FileInputStream("E:/nag5.txt");
    FileOutputStream fout=new FileOutputStream("E:/def.txt");
    DeflaterOutputStream out=new DeflaterOutputStream(fout);

    int i;
    while((i=fin.read())!=-1)
    {
        out.write((byte)i);
        out.flush();
    }
    fin.close();
    out.close();
}
catch(Exception e)
{
    System.out.println(e);
}
System.out.println("File Compressed successfully");
}
```

InflaterInputStream class:

The InflaterInputStream class is used to uncompress the file in the deflate compression format. It provides facility to the other uncompression filters, such as GZIPInputStream class.

uncompressing file using InflaterInputStream class

In this example, we are decompressing the compressed file def.txt into unfile.txt

Exp:-

```
import java.io.*;
import java.util.zip.*;
class UnCompress
{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("E:/def.txt");
            InflaterInputStream in=new InflaterInputStream(fin);
            FileOutputStream fout=new FileOutputStream("E:/Zip.txt");
            int i;
            while((i=in.read())!=-1)
            {
                fout.write((byte)i);
                fout.flush();
            }
            fin.close();
            fout.close();
        }
    }
}
```

```
        in.close();

    }

    catch(Exception e)

    {

        System.out.println(e);

    }

    System.out.println("File uncompressed successfully");

}

}
```

COREJAVA BY NAGARJUNA

Chapter6: Multithreading

Multitasking

Process-based Multitasking (Multiprocessing)

Thread-based Multitasking (Multiprocessing)

Thread

Life cycle of a Thread (Thread States)

Creating a thread

Thread Scheduler

Thread Priority

Sleeping a thread

Naming a Thread

Join() and isAlive() methods

Join():

Daemon Thread

Garbage Collection

finalize() method

Shutdown Hook

Serialization

Deserialization

Multithreading

Multithreading is a process of executing multiple threads simultaneously. So at this point we will ask ourselves what a thread is . A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process. So in short, Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system.

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, When you use a **word processor** you perform a many different tasks such as **printing, spell checking** and so on. Multithreaded software treats each process as a separate program.

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

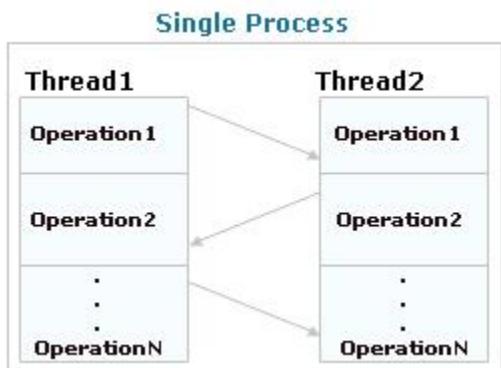
A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

I need to define another term related to threads: **process:** A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

In Java, the Java Virtual Machine (**JVM**) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.

For example, look at the diagram shown as:



In this diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

Multitasking:

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

Process-based Multitasking(Multiprocessing)

Thread-based Multitasking(Multithreading)

1)Process-based Multitasking (Multiprocessing):

- In process based multi tasking, several programs are executed at time, by the microprocessor.
- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

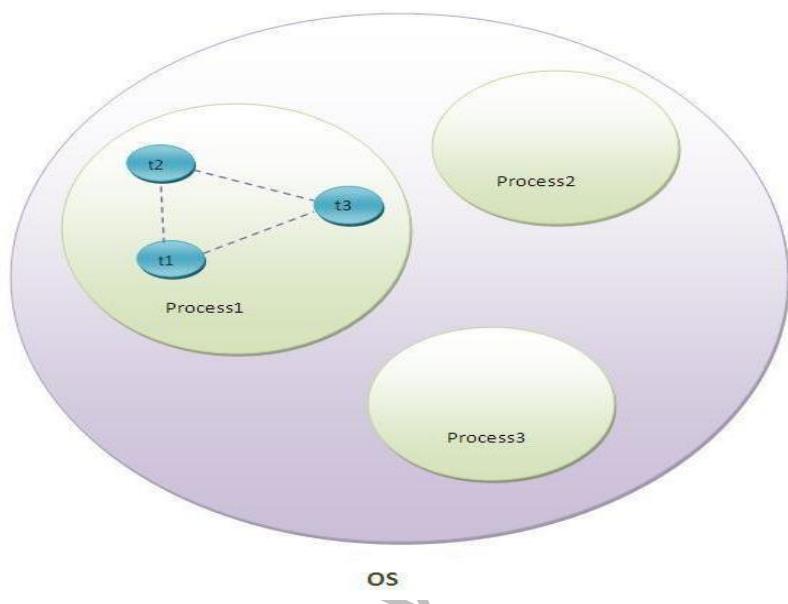
2)Thread-based Multitasking (Multithreading):

- In thread based multi tasking ,several parts are of the program is executed at a time, by the microprocessor.

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.
- Note: At least one process is required for each thread.

Thread:

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process.



above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time only one thread is executed.

Advantages of multithreading over multitasking :

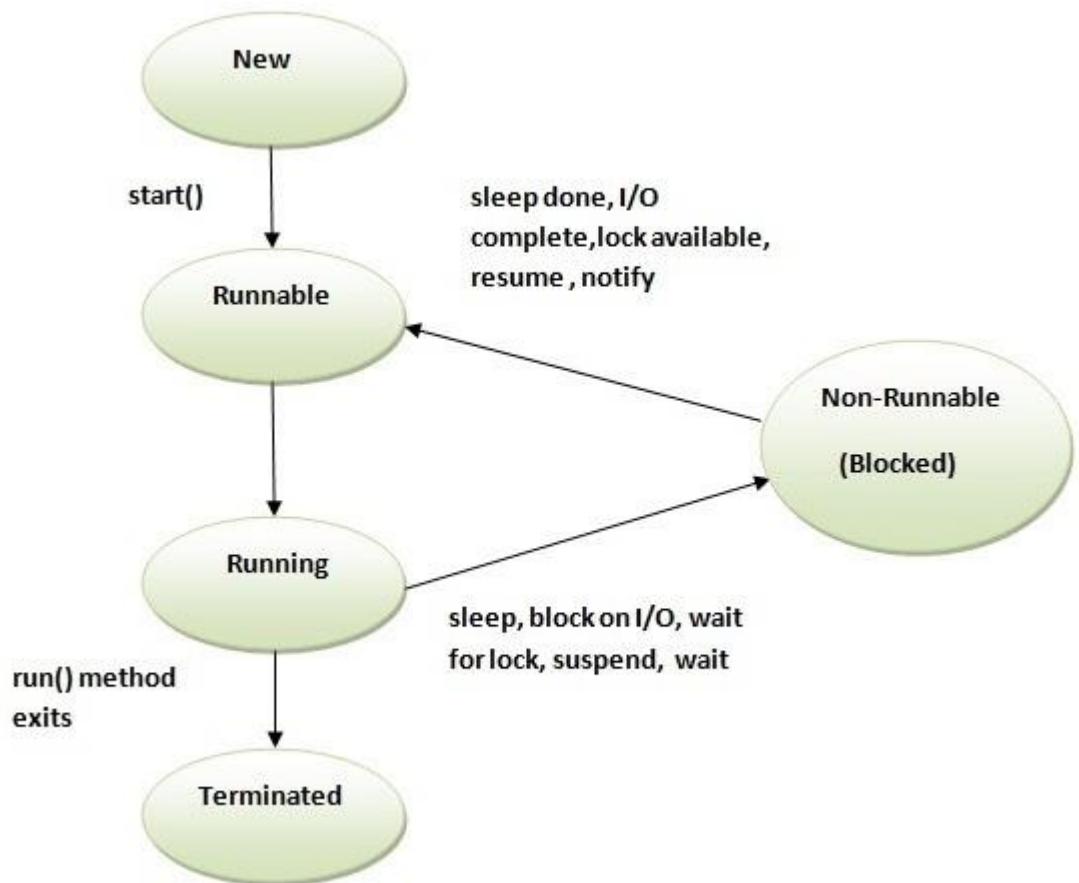
- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.

- Context switching between threads is usually less expensive than between processes.
- Cost of communication between threads is relatively low.

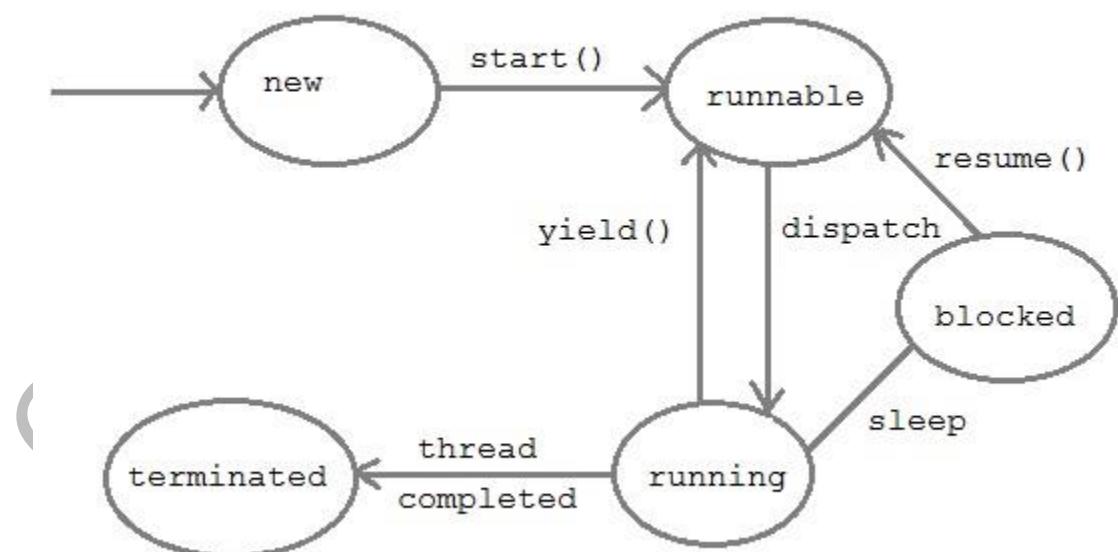
Life cycle of a Thread (Thread States):

A thread can be in one of the five states in the thread. According to sun, there is only 4 states new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads, we are explaining it in the 5 states. The life cycle of the thread is controlled by JVM. The thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



VA



1)New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2)Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3)Running

The thread is in running state if the thread scheduler has selected it.

4)Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5)Terminated

A thread is in terminated or dead state when its run() method exits.

Creating a thread:

we know that every java program there is a main thread available already. Apart from this main thread, we can also create our own threads in a program. In java there two ways to create the threads.

- 1) Extending the thread class.
- 2) Implementing the Runnable interface.

Main thread:

If we don't create any thread in your program, a thread called main thread is still created whenever we are writing the main method in our program, the main method is

automatically created. We can know the what is the present running thread by passing the thread reference to the **currentThread()** method.

Exp:-

```
class MyThreadMain extends Thread  
{  
    public static void main(String[] args)  
    {  
        Thread t=Thread.currentThread();  
        t.setName("MainThread");  
        System.out.println("Name of the Thread is:"+t);  
    }  
}
```

Output:

Name of the Thread is: Thread[MainThread,5,main]

Extending the thread class:

To create a Thread by a new class that **extends** thread class and create a instance of that class. That extending the class must be override run() method which is the entry point of new thread.

The user created thread always executed run method logic. So define the job for user thread we must be override run method in user defined class.

Main thread always executed main method logic.

To create a new user thread we must be call thread class start() method only.

Exp:-

```
class MyThread extends Thread
```

```
{  
public void run()  
{  
    System.out.println("biji");  
}  
public static void main(String[] args)  
{  
    MyThread mt=new MyThread();  
    mt.start();  
    System.out.println("h r u");  
}  
}
```

Note:

If multiple threads are waiting for getting the chance of execution then which thread will be get the chance first will be decided by the JVM. Hence we can't expect the exact output for the above program.

```
MyThread mt=new MyThread()  
mt.start();
```

In the above approach when we create a subclass object(MyThread) thread class object is also created by executing it's no argument constructor.

When we call the start() method by using subclass object reference a user thread is created and executed run() method.

Tikona Digital Networks > Creating a thread in Java - > multithreading in java home > Introduction to Java Multithreading > Multithreading in Java |

www.youtube.com/watch?v=O_Ojfq-OIpM

YouTube IN

Creating Threads

```

    graph TD
        A[Thread t = new Thread();] --> B[Just creates a Thread Object]
        B --> C[t.start();]
        C --> D["When start() is invoked, the immediate code that will be executed is from run method"]
        D --> E["public void run(){\n    // Code that should\n    // be executed by thread\n}"]
        E --> F["But the run in java.lang.Thread class has no link to Our application code."]
    
```

GUIDE

13:27 / 28:53

Multithreading in Java Part 1 | Introduction to Threads in Java | Java...
java9s 51 videos
Subscribe 14,408

16,749 144 4

1:35

Discover it here

Lungi Dance - Awesome Car by Sprite 20,991 views 1:35

Ad

Multithreading in Java Part 2 | Thread States | Java Tutorials by java9s 5,153 views

Exception Handling in Java - Checked and Unchecked by java9s 34,960 views 2:42

Show all downloads... 9:16 AM 12/14/2013

Tikona Digital Networks > Creating a thread in Java - > multithreading in java home > Introduction to Java Multithreading > Multithreading in Java |

www.youtube.com/watch?v=O_Ojfq-OIpM

YouTube IN

Creating Threads

```

    graph TD
        classDef1[class Downloader extends Thread{\n            private String url;\n            public Downloader(String url){\n                this.url = url;\n            }\n            public void run(){\n                FileDownloader fd = new FileDownloader();\n                fd.download(this.url);\n            }\n        }]
        classDef2[class FileDownload{\n            public File download(String url){\n                //code to download file\n                return file;\n            }\n        }]
        dt1((dt1)) --> fd1((fd))
        dt2((dt2)) --> fd2((fd))
        dt3((dt3)) --> fd3((fd))
        dt4((dt4)) --> fd4((fd))
        dt1.start();
        dt2.start();
        dt3.start();
        dt4.start();
    
```

GUIDE

15:50 / 28:53

Multithreading in Java Part 1 | Introduction to Threads in Java | Java...
java9s 51 videos
Subscribe 14,408

16,749 144 4

1:35

Discover it here

Lungi Dance - Awesome Car by Sprite 20,991 views 1:35

Ad

Multithreading in Java Part 2 | Thread States | Java Tutorials by java9s 5,153 views

Exception Handling in Java - Checked and Unchecked by java9s 34,960 views 2:42

Show all downloads... 9:31 AM 12/14/2013

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

A new thread starts(with new callstack).

The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run() method will run.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

Syntax: public void run(): is used to perform action

Implementing the Runnable interface:

Exp:-

```
class MyRunnable implements Runnable  
{  
    public void run()  
    {  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
        MyRunnable m1=new MyRunnable();  
        Thread t1 =new Thread(m1);  
    }  
}
```

```
t1.start();  
}  
}
```

Note:

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

(or)

In the above approach when we create an object for MyRunnable class, thread class object is not created, because MyRunnable is not a subclass of thread. So to call the start() method we should create thread class object explicitly by using Runnable parameterized constructor(thread(Runnable r)).

Case1: If we are not overriding the run() method

If we are not overriding the run() method. Then newly created user thread executed the thread class run() method.

Thread is an implemented class of Runnable interface. The run() method defined in Runnable interface.

Inside the thread class run() method implemented is empty implementation.

Exp:-

```
interface Runnable  
{  
    public abstract void run();
```

```
}

class Thread implements Runnable

{

public void run()

{

// empty

}

}

}
```

Exp:-

```
class MyThread extends Thread

{

public static void main(String[] args)

{

MyThread t=new My Thread();

t.start(); //it will execute the thread class run method

}

}
```

Case2: If we are calling the start() method twice

Once we created any thread if we are trying to restart the same thread once again then we will get the run time exception is an java.lang.IllegalThreadStateException.

Exp:-

```
class MultiStart2 extends Thread

{

public void run()

{

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

}

}

public static void main(String args[])
}
```

```
{  
    MultiStart2 t1=new MultiStart2();  
    t1.start();  
    t1.start();  
}  
}
```

Output:

```
running...Exception in thread "main"  
java.lang.IllegalThreadStateException  
at java.lang.Thread.start(Thread.java:638)  
at MultiStart2.main(MultiStart2.java:8)
```

Case3: we call run() method directly instead start() method

Each thread starts in a separate call stack.

Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

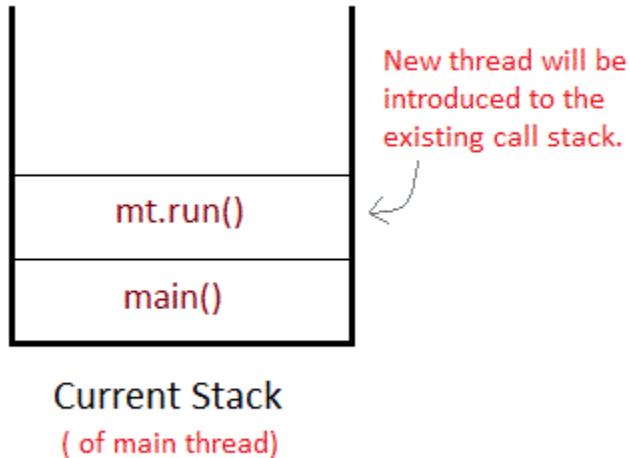
Exp:-

```
class DirectItRun extends Thread  
{  
    public void run()  
    {  
        System.out.println("running...");  
    }  
    public static void main(String args[])  
    {  
        DirectItRun mt=new DirectItRun();  
        mt.run(); //fine, but does not start a separate call stack  
    }  
}
```

```
}
```

Output:

running



The Thread won't be allocated a new call stack , and it will start running in the current call stack. That is the call stack of main thread. Hence Multithreading won't be there.

Problem if you direct call run() method

Exp:-

```
class DirectItRun extends Thread  
{  
    public void run()  
    {  
        for(int i=1;i<5;i++){  
            try  
            {  
                Thread.sleep(500);  
            }  
            catch(InterruptedException e)  
        }  
    }  
}
```

```
        {
            System.out.println(e);
        }

        System.out.println(i);
    }

}

public static void main(String args[])
{
    DirectItRun t1=new DirectItRun();

    DirectItRun t2=new DirectItRun();

    t1.run();

    t2.run();
}
```

Output:

```
1
2
3
4
1
2
3
4
```

As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

Case4: If we are overriding a start() method

Exp:-

```
class OverridingStartMethod extends Thread
{
    public void start()
    {
        System.out.println("start...");
    }

    public void run()
    {
        System.out.println("run...");
    }

    public static void main(String[] args)
    {
        OverridingStartMethod mt=new OverridingStartMethod();
        mt.start();
        System.out.println("main");
    }
}
```

Note: In the above case no new thread can be created by start() method. The start() method will be executed just like a normal method call and executed by main method.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Constructors:

1) Thread():

This constructor creates a new thread with the default characteristics.

Name of the Thread

Priority of the Thread

Thread group name.

Exp:-

```
class MyThread extends Thread  
{  
    public static void main(String[] args)  
    {  
        MyThread mt=new MyThread();  
        System.out.println(mt);  
    }  
}
```

Output: Thread[Thread-0,5,main]

Note:

In the above class `toString()` method is already overridden for meaning full representation. So we by pass any thread class object reference directly to S.O.Pln we will get the meaning full result.

2) Thread(String name):

This constructor creates the thread based on provided name thread object created.

Exp:-

```
class Con2ThreadStringname  
{  
    public static void main(String[] args)  
    {  
        Thread t=new Thread("nag");  
        System.out.println(t);  
    }  
}
```

```
}
```

Output: Thread[nag,5,main]

3) Thread(Runnable r):

This constructor creates the creates the Thread based on provided Runnable object reference and specified name a new thread object created.

Exp:-

```
class Con3ThreadRunnable
{
    public static void main(String[] args)
    {
        Runnable r=new Thread();
        Thread t=new Thread(r);
        System.out.println(t);
    }
}
```

Output: Thread[Thread-1,5,main]

4) Thread(Runnable r, String name):

This constructor is created the thread based on provide Runnable object reference and with specified name it will create new thread object.

Exp:-

```
class Con4ThreadRunnableStringName
{
    public static void main(String[] args)
    {
        Runnable r=new Thread();
        Thread t=new Thread(r,"arjun");
        System.out.println(t);
    }
}
```

```
}
```

Output: Thread[arjun,5,main]

5) Thread(ThreadGroup g, String s):

Based on the provided thread group and name ,the new thread object will be created. The ThreadGroup class is also present in java.lang package.

Exp:-

```
class Con5ThreadgroupStringName
{
    public static void main(String[] args)
    {
        ThreadGroup g=new ThreadGroup("nag");
        Thread t=new Thread(g,"arjun");
        System.out.println(t);
    }
}
```

Output: Thread[arjun,5,nag]

above Threadgroup is nag and threadname is arjun

6) Thread(ThreadGroup g, Runnable r, String name):

This constructor is created a new thread based on provided Runnable object reference and with specified ThreadGroup and thread name.

Exp:-

```
class Con6ThreadgroupRunnableStringName
{
    public static void main(String[] args)
    {
        ThreadGroup g=new ThreadGroup("nag");
        Runnable r=new Thread();
        Thread t=new Thread(g,r,"arjun");
    }
}
```

```
        System.out.println(t);
    }
}
```

Output: Thread[arjun,5,nag]

above ThreadGroup is nag and Threadname is arjun

Commonly used methods of Thread class:

public void run():

is used to perform action for a thread.

public void start():

starts the execution of the thread.JVM calls the run() method on the thread.

public void sleep(long milliseconds):

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void join():

waits for a thread to die.

public void join(long milliseconds):

waits for a thread to die for the specified milliseconds.

public int setPriority():

returns the priority of the thread.

public int setPriority(int priority):

changes the priority of the thread.

public String getName():

returns the name of the thread.

public void setName(String name):

changes the name of the thread.

public Thread currentThread():

returns the reference of currently executing thread.

public int getId():

returns the id of the thread.

public Thread.State getState():

returns the state of the thread.

public boolean isAlive():

tests if the thread is alive.

public void yield():

causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void suspend():

is used to suspend the thread(deprecated).

public void resume():

is used to resume the suspended thread(deprecated).

public void stop():

is used to stop the thread(deprecated).

public boolean isDaemon():

tests if the thread is a daemon thread.

public void setDaemon(boolean b):

marks the thread as daemon or user thread.

public void interrupt():

interrupts the thread.

public boolean isInterrupted():

tests if the thread has been interrupted.

public static boolean interrupted():

tests if the current thread has been interrupted.

The Thread Scheduler:

The thread scheduler is the part of the JVM that decides which thread should run.

There is no guarantee that which Runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses two types, to schedule the threads.

preemptive scheduling

time slicing scheduling

preemptive scheduling:

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.

time slicing scheduling:

Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

public static int MIN_PRIORITY

```
public static int NORM_PRIORITY
```

```
public static int MAX_PRIORITY
```

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Exp:-

```
//creating custom thread with user defined name and priority
```

```
class MyThread extends Thread
```

```
{
```

```
    public void run()
```

```
{
```

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

```
{
```

```
    System.out.println(getName()+"i:"+i);
```

```
}
```

```
}
```

```
}
```

```
public class ThreadNameAndPriority
```

```
{
```

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

```
{
```

```
    MyThread m1=new MyThread();
```

```
    MyThread m2=new MyThread();
```

```
    MyThread m3=new MyThread();
```

```
    System.out.println("m1 thread initialname and priority");
```

```
    System.out.println("m1 name:"+m1.getName());
```

```
    System.out.println("m1 priority:"+m1.getPriority());
```

```
    System.out.println();
```

```
System.out.println("m2 thread initialname and priority");
System.out.println("m2 name:"+m2.getName());
System.out.println("m2 priority:"+m2.getPriority());
System.out.println();
System.out.println("m3 thread initialname and priority");
System.out.println("m3 name:"+m3.getName());
System.out.println("m3 priority:"+m3.getPriority());
System.out.println();
m1.setName("child1");
m2.setName("child2");
m3.setName("child3");
m1.setPriority(6);
m2.setPriority(9);
m3.setPriority(10);
```

```
System.out.println("m1 thread changed initialname and priority");
System.out.println("m1 name:"+m1.getName());
System.out.println("m1 priority:"+m1.getPriority());
System.out.println();
System.out.println("m2 thread changed initialname and priority");
System.out.println("m2 name:"+m2.getName());
System.out.println("m2 priority:"+m2.getPriority());
System.out.println();
System.out.println("m3 thread changed initialname and priority");
System.out.println("m3 name:"+m3.getName());
System.out.println("m3 priority:"+m3.getPriority());
m1.start();
m2.start();
```

```
        m3.start();  
    }  
}
```

Output:

```
m1 thread initialname and priority  
m1 name:Thread-0  
m1 priority:5
```

```
m2 thread initialname and priority  
m2 name:Thread-1  
m2 priority:5
```

```
m3 thread initialname and priority  
m3 name:Thread-2  
m3 priority:5
```

```
m1 thread changed initialname and priority  
m1 name:child1  
m1 priority:6
```

```
m2 thread changed initialname and priority  
m2 name:child2  
m2 priority:9
```

```
m3 thread changed initialname and priority  
m3 name:child3  
m3 priority:10  
child1i:0
```

child2i:0

child3i:0

child1i:1

child3i:1

child2i:1

Sleeping a thread:

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method:

The Thread class provides two methods for sleeping a thread:

public static void sleep(long milliseconds) throws InterruptedException

public static void sleep(long milliseconds, int nanos) throws InterruptedException

Exp:-

```
class ThreadSleep extends Thread
{
    public void run()
    {
        System.out.println("nag");
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }
        System.out.println("arjun");
    }
}
```

```
public static void main(String args[])
{
    ThreadSleep t1=new ThreadSleep();
    ThreadSleep t2=new ThreadSleep();

    t1.start();
    t2.start();
}
```

Output:

```
nag  
nag  
arjun  
arjun
```

Note: In this above program two threads t1 and t2 are created. t1 starts first and after printing "nag" on console thread t1 goes to sleep for 500 mls. At the same time Thread t2 will start its process and print "nag" on console and then goes into sleep for 500 mls. Thread t1 wake up from sleep and print "arjun" on console similarly thread t2 will wake up from sleep and print "arjun" on console. So you will get output above output.

Naming a Thread (getName(),setName(),getId()):

All threads have names. By default, those names consist of the word Thread followed by a hyphen character (-), followed by an integer number starting at 0. You can introduce your own names by working with the setName() and getName() methods. Those methods make it possible to attach a name to a thread and retrieve a thread's current name, respectively. That name can be useful for debugging purposes.

The setName() method takes a String argument that identifies a thread. Similarly, the getName() method returns that name as a String.

Exp:-

```
class ThreadNaming extends Thread  
{  
    public void run()  
    {  
        System.out.println("running...");  
    }  
  
    public static void main(String args[])  
    {  
        ThreadNaming t1=new ThreadNaming();  
        ThreadNaming t2=new ThreadNaming();  
  
        System.out.println("Name of t1:"+t1.getName());  
        System.out.println("Name of t2:"+t2.getName());  
  
        t1.start();  
        t2.start();  
  
        t1.setName("nag");  
        t2.setName("arjun");  
  
        System.out.println("After changing name of t1:"+t1.getName());  
        System.out.println("After changing name of t2:"+t2.getName());  
    }  
}
```

Output:

Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:nag
After changing name of t1:arjun
running...
running...
After changing id of t1:9
After changing id of t2:10
currently active threads = 1

If we want to Know currently active threads:

Exp:-

```
class CurrentlyActiveThreads extends Thread
{
    public void run()
    {
        System.out.println("running...");
    }
    public static void main(String args[])
    {
        ThreadNaming t1=new ThreadNaming();
        ThreadNaming t2=new ThreadNaming();
        t1.start();
        t2.start();
        int count = Thread.activeCount();
        System.out.println("currently active threads = " + count);
    }
}
```

Output:

currently active threads = 3

running...

running...

// Here three thread is there that are two user defined threads and one main thread.

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax:

```
public static Thread currentThread()
```

Exp:-

```
class CurrentThreadExp extends Thread  
{  
    public void run()  
    {  
        System.out.println(Thread.currentThread().getName());  
    }  
  
    public static void main(String args[])  
    {  
        Thread t=Thread.currentThread();  
        String s=t.getName();  
        System.out.println("name:"+s);  
        CurrentThreadExp t1=new CurrentThreadExp();  
        CurrentThreadExp t2=new CurrentThreadExp();  
        t1.start();  
        t2.start();  
    }  
}
```

}

Output:

name: main

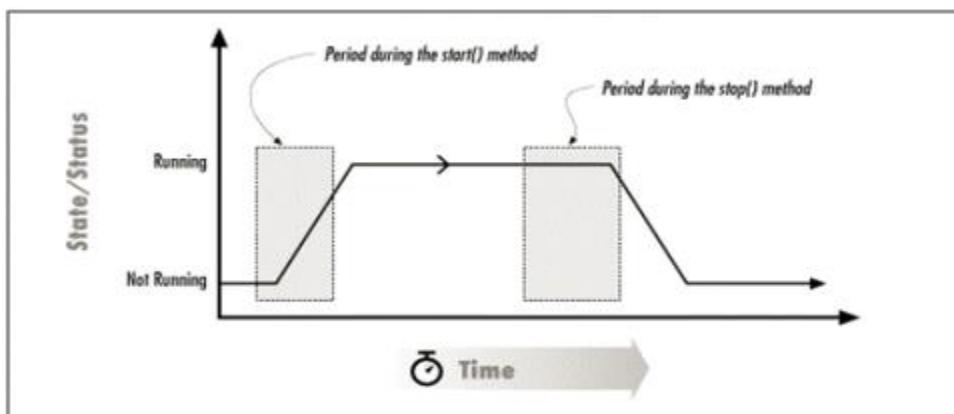
Thread-0

Thread-1

Join() and isAlive() methods:

isAlive():

isAlive() tests if this thread is alive. A thread is alive if it has been started and has not yet died. There is a transitional period from when a thread is running to when a thread is not running. After the run() method returns, there is a short period of time before the thread stops. If want to know if the start method of the thread has been called or if thread has been terminated, we must use isAlive() method. **This method is used to find out if a thread has actually been started and has yet not terminated.**



Join():

If we start a couple of thread to do a long calculation, we are then free to do other tasks. Assume that after some time we have completed all other secondary tasks and need to deal with the results of long calculation: we need to wait until the calculations are finished before continuing on to process the results.

Join()=sleep+isAlive()

When the join() method is called, the current thread will simply wait until the thread it is joining

with is no longer alive.

The isAlive() method simply returns the status of a thread, and the join() method simply

waits for a

certain status on the thread.

join(), isAlive(), and the Current Thread

The concept of a thread calling the isAlive() or the join() method on itself does not make sense. There is no reason to check if the current thread is alive since it would not be able to do anything about it if it were not alive. As a matter of fact, isAlive() can only return true when it checks the status of the thread calling it. If the thread were stopped during the isAlive() method, the isAlive() method would not be able to return. So a thread that calls the isAlive() method on itself will always receive true as the result.

The concept of a thread joining itself does not make sense, but let's examine what happens when one tries. It turns out that the join() method uses the isAlive() method to determine when to return from the join() method. In the current implementation, it also does not check to see if the thread is joining itself. In other words, the join() method returns when and only when the thread is no longer alive. This will have the effect of waiting forever.

Exp:-

```
( public class JoinAlive extends Thread {  
    public void run(){  
        try {
```

```
        System.out.println(Thread.currentThread().getName() + " is Started");

        Thread.sleep(2000);

        System.out.println(Thread.currentThread().getName() + " is Completed");

    }

    catch (InterruptedException ex)

    {

        System.out.println(ex);

    }

}

public static void main(String args[]) throws InterruptedException

{

    System.out.println(Thread.currentThread().getName() + " is Started");

    JoinAlive ja=new JoinAlive();

    System.out.println("thraed is started:"+ja.isAlive());

    ja.start();

    System.out.println("thraed is started:"+ja.isAlive());

    ja.join();

    System.out.println("thraed is join and complete its task and died");

    System.out.println("thraed is alive:"+ja.isAlive());
```

```
        System.out.println(Thread.currentThread().getName() + " is Completed");

    }

}
```

Output:

main is Started

thraed is started:false

thraed is started:true

Thread-0 is Started

Thread-0 is Completed

thraed is join and complete its task and died

thraed is alive:false

main is Completed

Note: If you look at above example, is started and then it creates another thread, whose name is "Thread-0" and started it. Since Thread-0 sleep for 2 seconds, it require at least 2 seconds to complete and in between main thread called join method on Thread-0 object. Because of join method, now main thread will wait until Thread-0 completes its operation or You can say main thread will join Thread-0. If you look on output, it confirms this theory.

Exp:-

```
class ThreadJoin extends Thread{

    public void run(){
```

```
for(int i=1;i<=5;i++){  
  
    try{  
  
        Thread.sleep(500);  
  
    }catch(Exception e){System.out.println(e);}  
  
    System.out.println(i);  
  
}  
  
}  
  
public static void main(String args[]){  
  
    ThreadJoin t1=new ThreadJoin();  
  
    ThreadJoin t2=new ThreadJoin();  
  
    ThreadJoin t3=new ThreadJoin();  
  
    System.out.println("t1 started:"+t1.isAlive());  
  
    t1.start();  
  
    System.out.println("t1 started:"+t1.isAlive());  
  
    try{  
  
        t1.join();  
  
    }catch(Exception e){System.out.println(e);}  
  
    System.out.println(", status = " + t1.isAlive());  
  
    System.out.println("t2 started:"+t2.isAlive());
```

```
t2.start();

System.out.println("t2 started:"+t2.isAlive());

try{

    t2.join();

}catch(Exception e){System.out.println(e);}

System.out.println(", status = " + t2.isAlive());

System.out.println("t3 started:"+t3.isAlive());

t3.start();

System.out.println("t3 started:"+t3.isAlive());

try{

    t3.join();

}catch(Exception e){System.out.println(e);}

System.out.println(", status = " + t3.isAlive());

}

}
```

Output:

t1 started:false

t1 started:true

2

3

4

5

, status = false

t2 started:false

t2 started:true

1

2

3

4

5

, status = false

t3 started:false

t3 started:true

1

2

3

4

, status = false

Daemon Thread:

There are two types of threads user thread and daemon thread. The daemon thread is a service provider thread(Garbage Collector). It provides services to the user thread. Its life depends on the user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

Points to remember for Daemon Thread:

It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

Its life depends on user threads.

It is a low priority thread.

Q) Why JVM terminates the daemon thread if there is no user thread remaining?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Daemon thread:

(The java.lang. Thread class provides two methods related to daemon thread

public void setDaemon(boolean status):

is used to mark the current thread as daemon thread or user thread.

public boolean isDaemon(): is used to check that current is daemon.

Exp:-

```
class DemonThreadExp extends Thread{  
  
    public void run(){  
  
        System.out.println("Name: "+Thread.currentThread().getName());  
  
        System.out.println("Daemon: "+Thread.currentThread().isDaemon());  
  
    }  
  
    public static void main(String[] args){  
  
        DemonThreadExp t1=new DemonThreadExp();  
  
        DemonThreadExp t2=new DemonThreadExp();  
  
        t1.setDaemon(true);  
  
        t1.start();  
  
        t2.start();  
  
        int count = Thread.activeCount();  
  
        System.out.println("currently active threads = " + count);  
  
    }  
}
```

Output:

currently active threads = 3

Name: Thread-1

Name: Thread-0

Daemon: false

Daemon: true

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw `IllegalThreadStateException`.

Exp:-

```
class DemonThreadExp extends Thread{  
  
    public void run(){  
  
        System.out.println("Name: "+Thread.currentThread().getName());  
  
        System.out.println("Daemon: "+Thread.currentThread().isDaemon());  
  
    }  
  
    public static void main(String[] args){  
  
        DemonThreadExp t1=new DemonThreadExp();  
  
        DemonThreadExp t2=new DemonThreadExp();  
  
        t1.setDaemon(true);  
  
        t1.start();  
  
        t2.start();  
  
        int count = Thread.activeCount();  
  
        System.out.println("currently active threads = " + count);  
    }  
}
```

}

}

Output:

Exception in thread "main" Name: Thread-0java.lang.IllegalThreadStateException

ThraedPooling:

Exp:-

```
import java.util.concurrent.ExecutorService;  
  
import java.util.concurrent.Executors;  
  
class WorkerThread implements Runnable {  
  
    private String message;  
  
    public WorkerThread(String s){  
  
        this.message=s;  
  
    }  
  
    public void run() {  
  
        System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);  
  
        processmessage();  
  
        System.out.println(Thread.currentThread().getName()+" (End)");  
  
    }  
}
```

```
private void processmessage() {  
  
    try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace();  
}  
  
}  
  
}  
  
public class ThreadPoolSimple {  
  
    public static void main(String[] args) {  
  
        ExecutorService executor = Executors.newFixedThreadPool(5);  
  
        for (int i = 0; i < 10; i++) {  
  
            Runnable worker = new WorkerThread("'" + i);  
  
            executor.execute(worker);  
  
        }  
  
        executor.shutdown();  
  
        while (!executor.isTerminated()) { }  
  
        System.out.println("Finished all threads");  
  
    }  
  
}
```

Output:

pool-1-thread-2 (Start) message = 1

pool-1-thread-4 (Start) message = 3

pool-1-thread-3 (Start) message = 2

pool-1-thread-1 (Start) message = 0

pool-1-thread-5 (Start) message = 4

pool-1-thread-2 (End)

pool-1-thread-2 (Start) message = 5

pool-1-thread-4 (End)

pool-1-thread-4 (Start) message = 6

pool-1-thread-3 (End)

pool-1-thread-3 (Start) message = 7

pool-1-thread-1 (End)

pool-1-thread-1 (Start) message = 8

pool-1-thread-5 (End)

pool-1-thread-5 (Start) message = 9

pool-1-thread-2 (End)

pool-1-thread-4 (End)

pool-1-thread-3 (End)

pool-1-thread-1 (End)

pool-1-thread-5 (End)

Finished all threads

Garbage Collection:

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. ▶

Advantage of Garbage Collection:

It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.

It is automatically done by the garbage collector so we don't need to make extra efforts.

How can an object be unreferenced:

There are many ways:

By nulling the reference

By assigning a reference to another

By anonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
```

```
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
```

```
Employee e2=new Employee();
```

```
e1=e2;//now the first object refered by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in System class as:

```
protected void finalize()  
{  
}
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc()  
{  
}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Exp:-

```
import java.util.*;

class GarbageCollection

{

    public static void main(String s[]) throws Exception

    {

        Runtime rs = Runtime.getRuntime();

        System.out.println("Free memory in JVM before Garbage Collection = "+rs.freeMemory());

        rs.gc();

        System.out.println("Free memory in JVM after Garbage Collection = "+rs.freeMemory());

    }

}
```

Output:

Free memory in JVM before Garbage Collection = 94399352

Free memory in JVM after Garbage Collection = 94270488

Note: Neither finalization nor garbage collection is guaranteed.

How to perform single task by multiple threads:

If you have to perform single task by many threads, have only one run() method

Program of performing single task by multiple threads

Exp:-

```
class Multi extends Thread{  
  
    public void run(){  
  
        System.out.println("task one");  
  
    }  
  
    public static void main(String args[]){  
  
        Multi t1=new Multi();  
  
        Multi t2=new Multi();  
  
        Multi t3=new Multi();  
  
        t1.start();  
  
        t2.start();  
  
        t3.start();  
  
    }  
}
```

Output:

task one

task one

task one

Program of performing single task by multiple threads:

Exp:-

```
class Multi3 implements Runnable

{

    public void run(){

        System.out.println("task one");

    }

    public static void main(String args[]){

        Thread t1 =new Thread(new Multi3());//passing anonymous object of Multi3
        class

        Thread t2 =new Thread(new Multi3());

        t1.start();

        t2.start();

    }

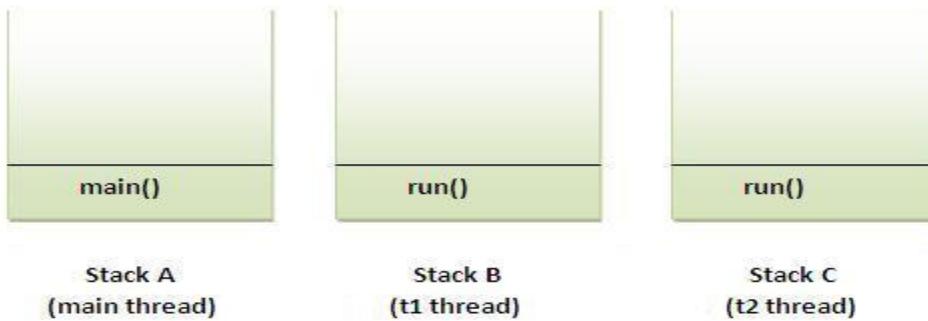
}
```

Output:

task one

task one

Note: Each thread run in a separate call stack.



How to perform multiple tasks by multiple threads (multitasking in multithreading):

If you have to perform multiple tasks by multiple threads, have multiple run() methods.

Program of performing two tasks by two threads

Exp:-

```
class Simple1 extends Thread{  
    public void run(){  
        System.out.println("task one");  
    }  
}
```

```
class Simple2 extends Thread{  
  
    public void run(){  
  
        System.out.println("task two");  
  
    }  
  
}  
  
class Test{  
  
    public static void main(String args[]){  
  
        Simple1 t1=new Simple1();  
  
        Simple2 t2=new Simple2();  
  
        t1.start();  
  
        t2.start();  
  
    }  
  
}
```

Output:

task one

task two

Exp:- Implementing Runnable

```
class Test{  
  
    public static void main(String args[]){
```

```
Runnable r1=new Runnable(){

    public void run(){

        System.out.println("task one");

    }

};

Runnable r2=new Runnable(){

    public void run(){

        System.out.println("task two");

    }

};

Thread t1=new Thread(r1);

Thread t2=new Thread(r1);

t1.start();

t2.start();

}

}
```

Output:

task one

task two

Shutdown Hook:

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

(or)

Shutdown hook allows performing some processing just before the java virtual machine shuts down. Shutdown hook is a simple thread which is registered with the JVM as a shutdown hook using the `addShutdownHook` method of the `java.lang.Runtime` class.

```
public void addShutdownHook(Thread hook)
```

When the virtual machine starts its shutdown sequence, it starts all registered shutdown hooks. The order of execution of these threads is not fixed as they run concurrently.

Preface:

Every Java Program can attach a shutdown hook to JVM, i.e. piece of instructions that JVM should execute before going down.

Problem:

A program may require to execute some pieces of instructions when application goes down. An application may go down because of several

When does the JVM shut down:

Because all of its threads have completed execution

Because of call to System.exit()

Because user hit CNTRL-C

System level shutdown or User Log-Off

Few scenarios where this requirement fits are:

Saving application state, e.g. when you exits from most of the IDEs, they remember the last view

Closing some database connections

Send a message to System Administrator that application is shutting down.

Solution:

Shutdown Hook comes to rescue in all such scenarios. Application attach a shutdown hook to their self, that JVM runs when application goes down.

Exp:-

```
public class JVMShutdownHookTest {  
  
    public static void main(String[] args) {  
  
        JVMShutdownHook jvmShutdownHook = new JVMShutdownHook();  
  
        Runtime.getRuntime().addShutdownHook(jvmShutdownHook);  
  
        System.out.println("JVM Shutdown Hook Registered.");  
  
        System.out.println("Pre exit.");  
  
        System.exit(0);  
    }  
}
```

```
System.out.println("Post exit.");  
}  
  
private static class JVMShutdownHook extends Thread {  
    ▶  
    public void run() {  
        System.out.println("JVM Shutdown Hook: Thread initiated.");  
    }  
}  
}
```

Output:

JVM Shutdown Hook Registered.

Pre exit.

JVM Shutdown Hook: Thread initiated.

In the above program, I manually call the `System.exit()` and make the JVM to shutdown. Once the `System.exit()` is invoked the phase I of shutdown sequence is initiated and that starts then registered thread ‘`jvmShutdownHook`’ and then halts the JVM.



Serialization

Serialization is a mechanism of writing the state of an object into a byte stream. It is mainly used in Hibernate, JPA, and EJB etc. The reverse operation of the serialization is called Deserialization. The String class and all the wrapper classes implements Serializable interface by default.

Advantage of Serialization

It is mainly used to travel object's state on the network.

About Serializable interface

Serializable is a marker interface (have no body). It is just used to "mark" Java classes which support a certain capability. It must be implemented by the class whose object you want to persist. Let's see the example given below:

Exp:-

```
import java.io.Serializable;  
  
public class Student implements Serializable{  
    int id;  
    String name;  
    public Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

ObjectOutputStream class:

An ObjectOutputStream is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Commonly used Constructors:

1) Public ObjectOutputStream (Output Stream out) throws IOException {}

Creates an ObjectOutputStream that writes to the specified Output Stream.

Commonly used Methods:

1) Public final void writeObject (Object obj) throws IOException {}

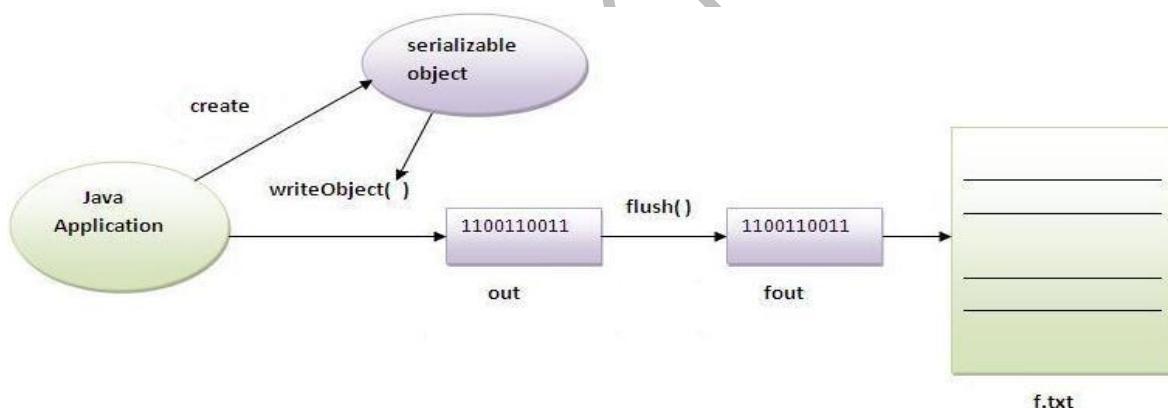
Write the specified object to the ObjectOutputStream.

2) Public void flush() throws IOException {}

flushes the current output stream.

Example of Serialization

In this example, we are going to serialize the object of Student class. The writeObject () method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.



Exp:-

```
import java.io.*;  
  
public class SerializeStudent implements Serializable  
{  
  
    int id;
```

```
String name;

public SerializeStudent(int id, String name)

{

this.id = id;

this.name = name;

}

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

{



SerializeStudent s1 =new SerializeStudent(101,"nag");

FileOutputStream fout=new FileOutputStream("D:/n.txt");

ObjectOutputStream out=new ObjectOutputStream(fout);

out.writeObject(s1);

out.flush();

System.out.println("success");

}

}


```

Output: Success

Deserialization:

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization.

ObjectInputStream class:

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Commonly used Constructors:

1) Public ObjectInputStream (InputStream in) throws IOException {}

Creates an ObjectInputStream that reads from the specified InputStream.

Commonly used Methods:

1) public final Object readObject() throws IOException, ClassNotFoundException{}

Reads an object from the input stream.

Exp:-

```
import java.io.*;

class Depersist{

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

ObjectInputStream in=new ObjectInputStream(new FileInputStream("D:/n.txt"));

SerializeStudent s=(SerializeStudent)in.readObject();

System.out.println(s.id+" "+s.name);

in.close();

}

}
```

Output: 101 nag

Serialization with Inheritance:

If a class implements Serializable then all its subclasses will also be Serializable.

Let's see the example given below:

Exp:-

```
import java.io.*;

class Person implements Serializable{

int id;

String name;
```

```
Person(int id, String name) {  
    this.id = id;  
    this.name = name;  
}  
}  
  
class Student extends Person{  
    String course;  
    int fee;  
    public Student(int id, String name, String course, int fee) {  
        super(id, name);  
        this.course=course;  
        this.fee=fee;  
    }  
}  
  
public class Persist  
{  
    public static void main(String[] args) throws Exception  
    {  
        Student s1= new Student(101,"nag","mca",30000);  
        FileOutputStream fos=new FileOutputStream("D:/n1.txt");  
        ObjectOutputStream oos=new ObjectOutputStream(fos);  
        oos.writeObject(s1);  
        oos.flush();  
        System.out.println("success");  
    }  
}
```

```
    }  
  
}
```

Output: success

Above we can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

Externalizable interface:

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

- 1) public void writeExternal(ObjectOutput out) throws IOException
- 2) public void readExternal(ObjectInput in) throws IOException

Serialization with Static data member:

Note: If there is any static data member in a class, it will not be serialized because static is related to class not to instance.

Exp:-

```
import java.io.*;  
  
public class SerializeStudent implements Serializable  
{  
    static int id;  
    String name;  
    public SerializeStudent(int id, String name)  
    {  
        this.id = id;  
        this.name = name;
```

```
}

public static void main(String args[])throws Exception{
    SerializeStudent s1 =new SerializeStudent(101,"nag");

    FileOutputStream fout=new FileOutputStream("D:/n.txt");

    ObjectOutputStream out=new ObjectOutputStream(fout);

    out.writeObject(s1);

    out.flush();

    System.out.println("success");

}

}
```

Output: Success

Deserialization:

Exp:-

```
import java.io.*;

class Depersist{

    public static void main(String args[])throws Exception{
        ObjectInputStream=in=new ObjectInputStream(new FileInputStream("D:/n1.txt"));

        Student s=(Student)in.readObject();

        System.out.println(s.id+" "+s.name);

        in.close();

    }

}
```

Output: 0, nag

Synchronization

Threads commonly share the same memory space area, that's why they can share the resources. Threads commonly communicate by sharing access to fields and the objects reference fields refer to. This communication type is extremely efficient, but makes two kinds of problems: thread interference and memory consistency errors. By the synchronization tool we can avoid this problem. In other words, There is very critical situation where we want only one thread at a time has to access a shared resources. For example, suppose two people each have a checkbook for a single account same as like two different threads are accessing the same account data.

Synchronization is a process of sharing common resources among multiple threads so that only one thread can access that resource at a time. The resources may be printer, a file, database connection etc. In java we use synchronized key word to achieve synchronization. We create synchronized methods or synchronized blocks to block the and the those code which needs to be thread safe.

Lock

This term refers to the access granted to a particular thread that can access the shared resources. At any given time, only one thread can hold the lock and thereby have access to the shared resource. Every object in Java has build-in lock that only comes in action when the object has synchronized method code. By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the object lock.

Since there is one lock per object, if one thread has acquired the lock, no other thread can acquire the lock until the lock is not released by first thread. Acquire the lock means the thread currently in synchronized method and released the lock means exits the synchronized method. Remember the following points related to lock and synchronization:

- ❖ Only methods (or blocks) can be synchronized, Classes and variable cannot be.
- ❖ Each object has just one lock.
- ❖ All methods in a class need not to be synchronized. A class can have both synchronized and non-synchronized methods.
- ❖ If two threads wants to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method then only one thread can execute the method at a time.
- ❖ If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods. If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
- ❖ If a thread goes to sleep, it holds any locks it has it doesn't release them.
- ❖ A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again.
- ❖ You can synchronize a block of code rather than a method.
- ❖ Constructors cannot be synchronized

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1.Mutual Exclusive

Synchronized method.
Synchronized block.
static synchronization.

2.Cooperation (Inter-thread communication)

Mutual Exclusive

problem without Synchronization

Exp:-

```
class Table
{
    void printTable(int n)
    { //method not synchronized
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
            try
            {
                Thread.sleep(400);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

```
}

}

}

}

class MyThread1 extends Thread

{

Table t;

MyThread1(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(5);

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t)

{

this.t=t;
```

```
}

public void run()

{

t.printTable(100);

}

}
```

```
class WithOutSync

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}
```

(Output:

```
100

5

10
```

200

15

300

20

400

25

500

1.Synchronized Methods

If any method is specified with the keyword synchronized then this method of an object is only executed by one thread at a time. A any thread want to execute the synchronized method, firstly it has to obtain the objects lock. Acquire the method is simply by calling the method. If the lock is already held by another thread, then calling thread has to wait.

- ❖ If you declare any method as synchronized, it is known as synchronized method.
- ❖ Synchronized method is used to lock an object for any shared resource.
- ❖ When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the method returns.

Using the Synchronization for Method:

Exp:-

```
class Table
```

```
{
```

```
    synchronized void printTable(int n)
```

```
    { //method not synchronized
```

```
for(int i=1;i<=5;i++)  
{  
    System.out.println(n*i);  
    try  
    {  
        Thread.sleep(400);  
    }  
    catch(Exception e)  
    {  
        System.out.println(e);  
    }  
}  
}  
}  
}
```

```
class MyThread1 extends Thread  
{  
    Table t;  
    MyThread1(Table t)  
    {  
        this.t=t;  
    }
```

```
public void run()

{

t.printTable(5);

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(100);

}

}

class WithSync

{

public static void main(String args[])

{

Table obj = new Table();//only one object
```

```
MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}
```



Output:

```
5
10
15
20
25
100
200
300
400
500
```

2.Synchronized block



- ❖ Synchronized block can be used to perform synchronization on any specific resource of the method.
- ❖ Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- ❖ If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- ❖ Synchronized block is used to lock an object for any shared resource.
- ❖ Scope of synchronized block is smaller than the method.

Syntax:-

```
synchronized (object reference expression)
```

```
{
```

```
//code block
```

```
}
```

Exp:-

```
class Table
```

```
{
```

```
void printTable(int n)
```

```
{
```

```
    synchronized(this)
```

```
        {//synchronized block
```

```
            for(int i=1;i<=5;i++)
```

```
{
```

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

```
            try
```

```
{
```

```
                Thread.sleep(400);
```

```
}
```

```
        catch(Exception e)

        {

            System.out.println(e);

        }

    }

}

//end of the method

}

class MyThread1 extends Thread

{

    Table t;

    MyThread1(Table t)

    {

        this.t=t;

    }

    public void run()

    {

        t.printTable(5);

    }

}

class MyThread2 extends Thread

{
```

```
Table t;

MyThread2(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(100);

}

}

class SynBlock

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}


```

Output:

5
10
15
20
25
100
200
300
400
500



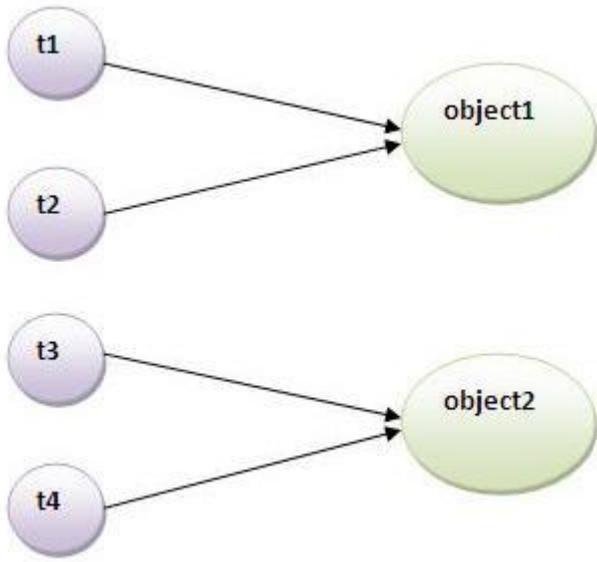
Static synchronization

When Synchronization is applied on a static Member or a static block, the lock is performed on the Class and not on the Object, while in the case of a Non-static block/member, lock is applied on the Object and not on class.

Syn:-

```
class MyClass {  
...  
public synchronized static someMethod() {  
}  
}
```





Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Exp:-

```

class Table
{
    synchronized static void printTable(int n)
    {
        for(int i=1;i<=3;i++)
        {
            System.out.println(n*i);
            try
            {
                Thread.sleep(400);
            }
            catch(Exception e)
            {

```

```
System.out.println(e);
}

}

}

}

class MyThread1 extends Thread
{
public void run()
{
Table.printTable(1);
}

class MyThread2 extends Thread
{
public void run()
{
Table.printTable(10);
}

class MyThread3 extends Thread
{
public void run()
{
Table.printTable(100);
}

class MyThread4 extends Thread
{
public void run()
{
Table.printTable(1000);
}
```

```
}

class StaticSyn
{
    public static void main(String t[])
    {
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        MyThread3 t3=new MyThread3();
        MyThread4 t4=new MyThread4();
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

It's possible to synchronize a static method. When this occurs, a lock is obtained for the class itself. This is demonstrated by the static hello() method in the SyncExample class below. When we create a synchronized block in a static method, we need to synchronize on an object, so what object should we synchronize on? We can synchronize on the Class object that represents the class that is being synchronized. This is demonstrated in the static goodbye() method of SyncExample. We synchronize on SyncExample. Class.

Exp:-

```
public class StaticSyncExample
{
    public static void main(String[] args)
    {
        hello();
        goodbye();
    }

    public static synchronized void hello()
```

```
{  
    System.out.println("hello");  
}  
public static void goodbye()  
{  
    synchronized (StaticSyncExample.class)  
    {  
        System.out.println("goodbye");  
    }  
}
```

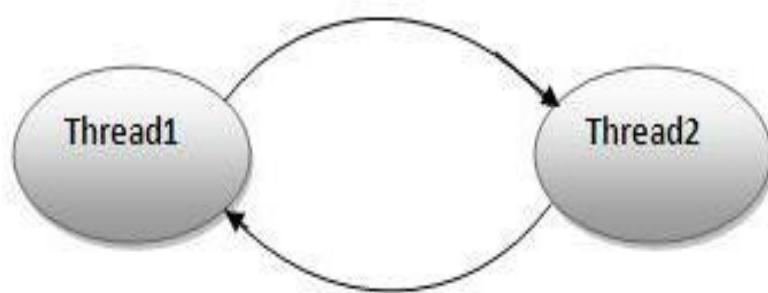
Note that in this example, the synchronized block synchronizes on the SyncExample Class object. When using a synchronized block, we can also synchronize on another object. That object will be locked while the code in the synchronized block is being executed.

Deadlock:

When two threads or processes are waiting for each other to release the resource or object they holds and so are blocked forever. This situation is called deadlock. For example if one thread is holding the lock on some object that the other thread is waiting for and the other thread is holding lock on some object the first one is waiting for then they both will wait for each other to release the object they need to finish their operation but no one will release the hold object and so they will wait for each other forever.

(or)

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Exp:-

```
class Pen{}  
class Paper{}  
public class WriteDeadLock  
{  
  
    public static void main(String[] args)  
{  
        final Pen pn =new Pen();  
        final Paper pr =new Paper();  
        Thread t1 = new Thread(){  
            public void run()  
            {  
                synchronized(pn)  
                {  
                    System.out.println("Thread1 is holding Pen");  
                    try{  
                        Thread.sleep(1000);  
                    }catch(InterruptedException e){}  
                    synchronized(pr)  
                    { System.out.println("Requesting for Paper"); }  
                }  
            }  
        };  
        t1.start();  
        Thread t2 = new Thread(){  
            public void run()  
            {  
                synchronized(pr)  
                {  
                    System.out.println("Thread2 is holding Paper");  
                    try{  
                        Thread.sleep(1000);  
                    }catch(InterruptedException e){}  
                    synchronized(pn)  
                    { System.out.println("Requesting for Pen"); }  
                }  
            }  
        };  
        t2.start();  
    }  
}
```

```
    }
}
};

Thread t2 = new Thread(){
public void run()
{
synchronized(pr)
{
System.out.println("Thread2 is holding Paper");
try{
Thread.sleep(1000);
}
catch(InterruptedException e){}
synchronized(pn)
{
System.out.println("requesting for Pen"); }

}
}
};

t1.start();
t2.start();
}
}

Output:
```

Thread1 is holding Pen
Thread2 is holding Paper

Inter-thread communication

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

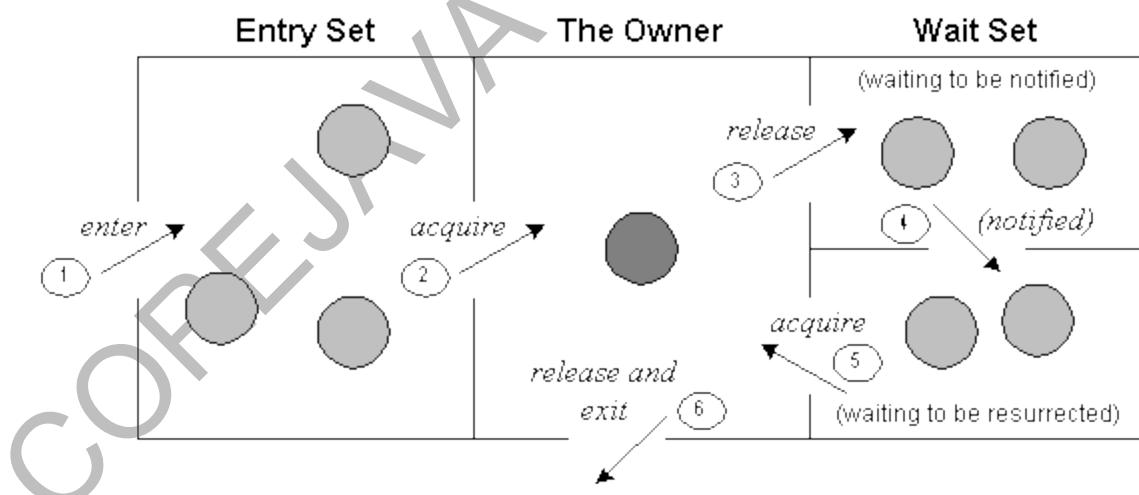
wait()
notify()
notifyAll()

wait(): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()**.

notify(): wakes up the first thread that called **wait()** on the same object.

notifyAll(): wakes up all the threads that called **wait()** on the same object. The highest priority thread will run first.

Process of inter-thread communication



1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call **wait()** method on the object.
Otherwise it releases the lock and exits.

4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class

It is because they are related to lock and object has a lock.

wait()

1. wait() method releases the lock lock.
2. is the method of Object class
3. is the non-static method
4. should be notified by notify() or sleep is notifyAll() methods

sleep()

- sleep() method doesn't release the
is the method of Thread class
is the static method
after the specified amount of time,
completed.

Exp:-

```
class Customer
{
    int amount=10000;
    synchronized void withdraw(int amount)
    {
        System.out.println("going to withdraw...");
        if(this.amount<amount)
        {
            System.out.println("Less balance; waiting for deposit...");
            try{wait();}catch(Exception e){}
        }
        this.amount-=amount;
        System.out.println("withdraw completed...");
    }
}
```

```
}

synchronized void deposit(int amount)
{
    System.out.println("going to deposit...");
    this.amount+=amount;
    System.out.println("deposit completed... ");
    notify();
}
}

class InterThreadComm{
public static void main(String args[])
{
    final Customer c=new Customer();
    new Thread()
    {
        public void run()
        {
            c.withdraw(15000);
        }
    }.start();
    new Thread()
    {
        public void run()
        {
            c.deposit(10000);
        }
    }.start();
}
}
```

Output:

going to withdraw...
Less balance; waiting for deposit...

going to deposit...
deposit completed...
withdraw completed...

Difference between sleep() and wait():

sleep() is a method which is used to hold the process for few seconds or the time you wanted but in case of wait() method thread goes in waiting state and it won't come back automatically until we call the notify() or notifyAll().

The major difference is that wait() releases the lock or monitor while sleep() doesn't releases any lock or monitor while waiting. Wait is used for inter-thread communication while sleep is used to introduce pause on execution, generally.

Thread.sleep() sends the current thread into the "Not Runnable" state for some amount of time. The thread keeps the monitors it has acquired — i.e. if the thread is currently in a synchronized block or method no other thread can enter this block or method. If another thread calls t.interrupt() it will wake up the sleeping thread. Note that sleep is a static method, which means that it always affects the current thread (the one that is executing the sleep method). A common mistake is to call t.sleep() where t is a different thread; even then, it is the current thread that will sleep, not the t thread.

object.wait() sends the current thread into the "Not Runnable" state, like sleep(), but with a twist. Wait is called on an object, not a thread; we call this object the "lock object." Before lock.wait() is called, the current thread must synchronize on the lock object; wait() then releases this lock, and adds the thread to the "wait list" associated with the lock. Later, another thread can synchronize on the same lock object and call lock.notify(). This wakes up the original, waiting thread. Basically, wait()/notify() is like sleep()/interrupt(), only the active thread does not need a direct pointer to the sleeping thread, but only to the shared lock object.

```
synchronized(LOCK) {  
    Thread.sleep(1000); // LOCK is held
```

```
}

synchronized(LOCK) {
    LOCK.wait(); // LOCK is not held
}
```

Let categorize all above points :

Call on:

wait(): Call on an object; current thread must synchronize on the lock object.

sleep(): Call on a Thread; always currently executing thread.

Synchronized:

wait(): when synchronized multiple threads access same Object one by one.

sleep(): when synchronized multiple threads wait for sleep over of sleeping thread.

Hold lock:

wait(): release the lock for other objects to have chance to execute.

sleep(): keep lock for at least t times if timeout specified or somebody interrupt.

Wake-up condition:

wait(): until call notify(), notifyAll() from object

sleep(): until at least time expire or call interrupt().

Usage:

sleep(): for time-synchronization and;

wait(): for multi-thread-synchronization.

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behavior and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

```
public void interrupt()  
public static boolean interrupted()  
public boolean isInterrupted()
```

interrupting a thread that stops working

Above example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

Exp:-

```
class Interupt extends Thread  
{  
    public void run()  
    {  
        try  
        {  
            Thread.sleep(1000);  
            System.out.println("task");  
        }  
        catch(InterruptedException e)  
        {
```

```
        throw new RuntimeException("Thread interrupted..."+e);
    }
}

public static void main(String args[]){
    Interupt t1=new Interupt();
    t1.start();
    try
    {
        t1.interrupt();
    }
    catch(Exception e)
    {
        System.out.println("Exception handled "+e);
    }
}
}
```

Output:

```
Exception in thread Thread-0
java.lang.RuntimeException: Thread interrupted...
java.lang. InterruptedException: sleep interrupted
at Interupt.run(Interupt.java:12)
```

interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

Exp:-

```
class InteruptWorking extends Thread
{
    public void run()
{
```

```
try
{
    Thread.sleep(1000);
    System.out.println("task");
}
catch(InterruptedException e)
{
    System.out.println("Exception handled "+e);
}
System.out.println("thread is running...");
}

public static void main(String args[])
{
    InteruptWorking t1=new InteruptWorking();
    t1.start();
    t1.interrupt();
}
```

Output:

```
Exception handled
java.lang. InterruptedException: sleep interrupted
thread is running...
```

Starvation and LiveLock in Java

Starvation

Suppose there are multiple threads in a Java process, all needs to get lock on a particular object. Assume the first thread holds lock on the shared object for a long time. For this entire time others threads are waiting. Suppose the long duration thread is invoking very frequently. So other threads will be blocked from accessing the shared object. This situation is referred as starvation in multi-threading.

LiveLock

It is a situation in multi-threading when two or more threads are unable to make any progress and they are not blocked each other. These threads are responding to each other to finish the tasks, but no progress is going to happen.

COREJAVA BY NAGARJUNA

Collection's

collection:

A collection is java objects that is used for storing a group of homogeneous and heterogeneous, unique and duplicate objects without size limitation. Collection object is also called as container object.

In how many ways we can store data in JVM:

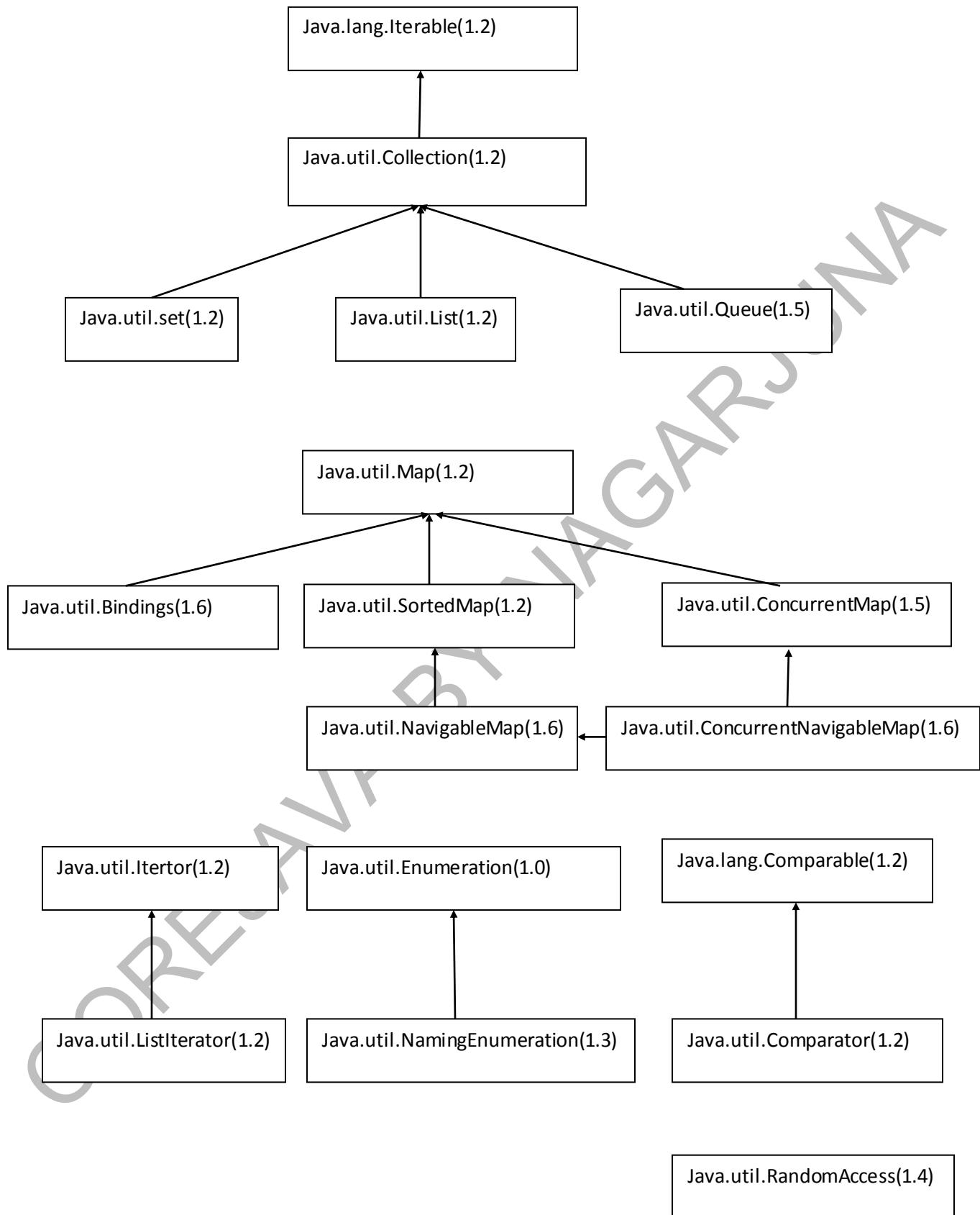
We can store data in four ways in JVM temporally.

- 1) By using variable
- 2) By using array Object
- 3) By using class Object
- 4) By using collection Object

Why do we need collection when we have array for collecting object:

Array has two limitations

- 1) it allows only homogeneous objects
- 2) we can't change its size after it is created
 - The first problem we can solve by creating `java.lang.Object` class
 - To second problem size limitation problem SUN has defined collection API with well defined algorithm and data structure.
 - If collection API is not given by SUN every java developer must develop code for `Object[]` size limitation problem. Then every developer in every project develop the same code with their own class names and method names.
 - This leads to lot of maintenance issues, and also when developer change the company, should learn new API to perform same task.



Collection interface methods:

1) boolean add(Object obj):

This method is used to insert an element into the collection object.

2) boolean addAll(collection c):

This method is used to insert one collection object elements into another collection object.

Exp:-

```
ArrayList al=new ArrayList();
LinkedList ll=new LinkedList();
al.add("nag");
al.add("arjun");
ll.add(al);
```

3) public boolean remove(object ele):

Used to remove an element from the collection object.

4) boolean removeAll(collection c):

remove all the elements of specified collection object.

5) boolean retainAll(collection c):

This method is used to remove all the elements except those are specified in the collection object.

6) void clear():

This is used to removes the total no of elements from the collection.

7) int size():

It is used to return the total no of elements present in the collection.

8) public boolean contains():

It is used to check an element (or) object is present in the collection.

9) boolean containsAll(collection c):

It is used to check the specified collection object in their are not in the another collection object.

10) public Iterator iterator():

This method returns an iterator object reference is used to retrieve the elements from collection object.

Exp:-

```
Iterator itr=c.iterator()
```

Here c is collection object

11) Object[] to Array():

This is used to represent collection object in the form of an object array.

12) boolean equals(Object obj):

13) int hashCode():

14) boolean isEmpty():

List Interface:

List is the child interface of collection. If we want to represent a group of object as a single entity where duplicate are allowed insertion order id followed we should go for list.

We can maintain insertion order and we can differentiate duplicate objects by means of an index. Hence index place very important role in list.

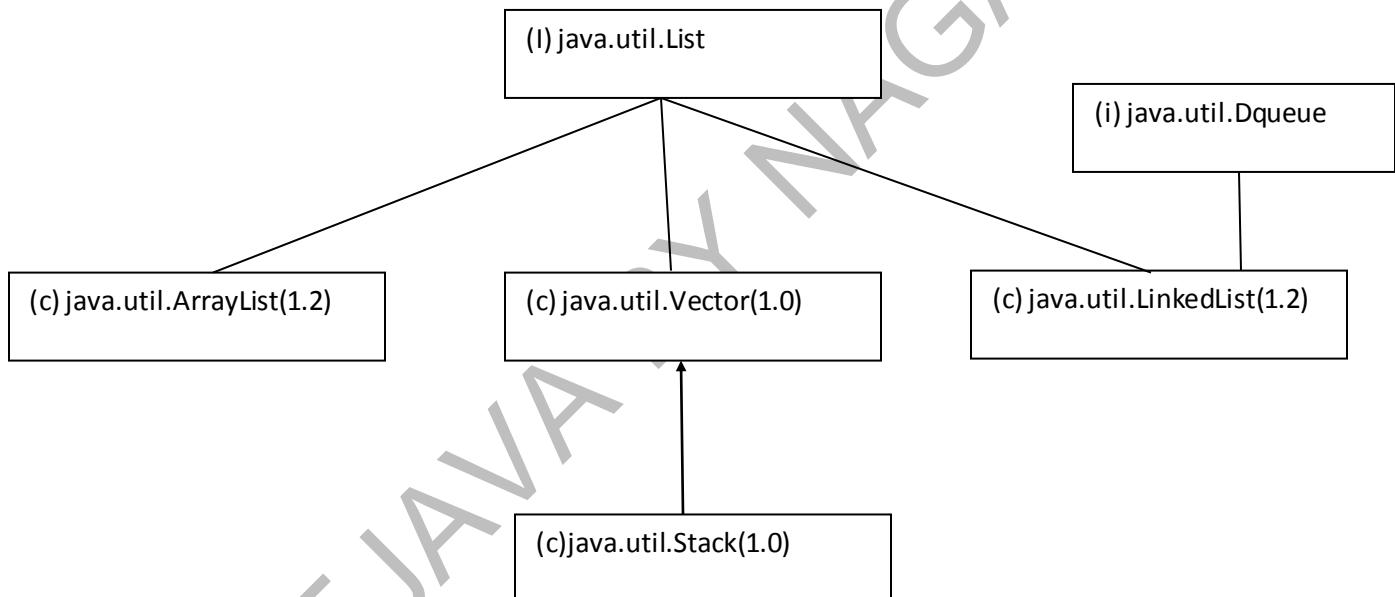
The following methods are specified in list interface:

1) boolean add(int index, Object obj)

2) boolean addAll(int index, collection c)

- 3) object remove(int index)
- 4) object get(int index)
- 5) int indexOf(object o)
- 6) int lastIndexOf(object o)
- 7) object set(int index, object o)
- 8) ListIterator ListIterator()

List interface and it's Implementations:



Above ----- means Implementation

-----> means extends

ArrayList:

`ArrayList` is an implemented class of list interface.

`ArrayList` object is dynamically resizable

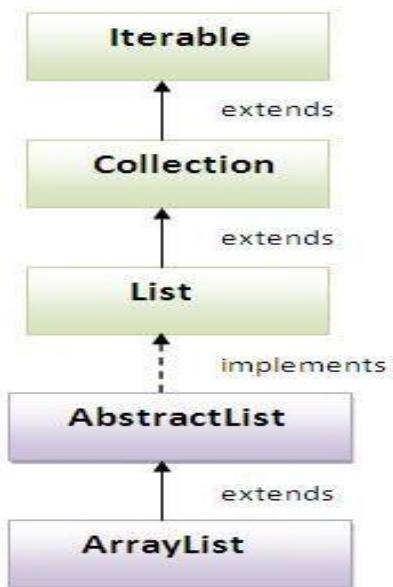
`ArrayList` object maintain the duplicate elements

ArrayList object maintain the elements in the list Insertion order

ArrayList object also accept the homogenous and heterogeneous elements.

ArrayList object also maintain the elements based on index

ArrayList object accept null pointer constant any number of times.



ArrayList class Constructor:

ArrayList():

This constructor creates an empty `ArrayList` object with default initial capacity is 10. If the array list object is reaches it's max capacity then a new arraylist object will be created.

```
new capacity=(current capacity*3)/2+1
```

```
ArrayList al=new ArrayList()
```

ArrayList(int initial capacity):

It creates an empty arraylist object with the specified initial capacity if arraylist object reaches its max capacity then a new arraylist object will be created.

```
ArrayList al=new ArrayList(100)
```

ArrayList(collection c): This constructor creates an equivalent arraylist object for the given collection.

```
collection c=new vector();
ArrayList al=new ArrayList(c);
```

Exp:-

```
import java.util.ArrayList;
public class ArrayDemo
{
    public static void main(String[] args)
    {
        ArrayList al=new ArrayList();
        //System.out.println("initial capacity:"+al.length());
        System.out.println("initial size:"+al.size());
        al.add("nag");
        al.add("arjun");
        al.add("raj");
        al.add("sunny");
        al.add("bujji");
        al.add(90);
        al.add(80);
        al.add(90);
        al.add("nag");
        al.add("raj");
        al.add('a');
```

```
//System.out.println("ArrayList capacity:"+al.capacity());  
  
System.out.println("ArrayList size:"+al.size());  
  
System.out.println(al);  
  
System.out.println(".....");  
  
//al.remove(20);  
  
al.remove("nag");  
  
al.remove("arjun");  
  
System.out.println(al);  
  
}  
  
}
```

Note:

If you are bypassing any object reference directly to S.O.Pln statement automatically internally the `toString()` method is invoked. Object class `toString()` method always returns "class name @hexadecimal format of an hashCode".

to get the meaningful result it is highly recommended to override the `toString()` method in the user defined classes.

In the String class, String Buffer class, String Builder class, all the wrapper classes(Byte, Short, Integer, Long, Double, Float, Char, Boolean), all the collection (List, set, map) ,In throwable classes in exception and error classes, In thread classes already `toString()` was overridden for a meaningful representation.

Advantages:

ArrayList objects are Serializable objects why because ArrayList implements Serializable interface. generally we can use collection to hold objects and transfer the objects data across the network that process is called Serialization.

ArrayList objects are Cloneable Objects why because ArrayList implements Cloneable interface. To create the exactly duplicate objects.

ArrayList Object implements Random Access interfaces interface for frequent retrieval operation.

Hence if our frequent operation is retrieval operation ArrayList is the best choice.

Disadvantage:

ArrayList is not best suitable for frequent insertion & deletion operation as it requires internally lot of shifting operations.

Vector:

Vector class is an implemented class of list interface

Vector object is dynamically resizable

Vector object maintain the duplicate elements

Vector object maintain the elements in the list Insertion order

Vector object also accept the homogenous and heterogeneous elements.

Vector object also maintain the elements based on index

Vector object accept null pointer constant any number of times.

Constructors:

Vector():

This constructor creates an empty vector object with default initial capacity is 10. If the vector object reaches its max capacity then a new vector object will be created.

new capacity = 2 * current capacity

Vector(int i):

It creates an empty arraylist object with the specified initial capacity if arraylist object reaches its max capacity then a new arraylist object will be created.

```
vector v=new vector(40);
```

Vector (collection c):

Creates a vector containing the elements of the specified collection c.

Exp:-

```
import java.util.Vector;  
  
class VectorDemo  
  
{  
  
    public static void main(String[] args)  
  
    {  
  
        Vector v=new Vector();  
  
        System.out.println("initial capacity:"+v.capacity());  
  
        System.out.println("size:"+v.size());  
  
        v.add("nag");  
  
        v.add("arjun");  
  
        v.add("raj");  
  
        v.add(10);  
  
        v.add(20);  
  
        v.add(10);  
  
        v.add(10);  
  
        v.add(20);  
  
        v.add(10);  
  
        v.add('a');  
  
        v.add("nag");  
  
        System.out.println(v);  
  
        v.remove(0);  
  
        System.out.println("capacity:"+ v.capacity());  
  
        System.out.println("size:"+ v.size());
```

```
        System.out.println(v);

    }

}
```

Vector class Specific methods:

For Adding Objects:

- 1) void add(Object obj) // from collection
- 2) void add(int index, Object o) //from list
- 3) void addElement(Object obj) // vector

For removing elements:

- 1) Object remove(Object o) // Collection
- 2) Object remove(int index) //list
- 3) Object removeElement(Object o) //vector
- 4) Object removeElementAt(int index) //vector
- 5) Object clear() //Collection
- 6) removeAllElements() //vector

For retrieving Objects:

- 1) Object get(int index) //list
- 2) Object elementAt(int index) //vector
- 3) Object firstElement() //vector
- 4) Object lastElement() // vector

Other common methods:

- 1) int capacity() //vector
- 2) int size() //collection
- 2) Enumeration elements() // vector

Advantages:

Vector objects are Serializable objects why because Vector implements Serializable interface. generally we can use collection to hold objects and transfer the objects data across the network that process is called Serialization.

Vector objects are Cloneable Objects why because Vector implements Cloneable interface. To create the exactly duplicate objects.

Vector Object implements Random Access interfaces interface for frequent retrieval operation.

Hence if our frequent operation is retrieval operation Vector is the best choice.

Disadvantage:

Vector is not best suitable for frequent insertion & deletion operation as it requires internally lot of shifting operations.

The main difference between the ArrayList and Vector classes:

The main difference between ArrayList class and vector class thread safe.

ArrayList not thread safe (not a synchronized methods)

Vector is thread safe (synchronized methods)

ArrayList

- 1) ArrayList class contains non-synchronized Methods. hence multiple threads can operate simultaneously an ArrayList object.
- 2) ArrayList is not Thread safe
- 3) Performance is high
- 4) It is non-legacy class. It is introduced in java 1.2 version.

Vector

- 1) vector class contains synchronized methods. Hence only one thread can allowed vector object.
- 2) vector is thread safe
- 3) Performance is low
- 4) It is legacy class. It is introduced in 1.0 version.

Add one collection object into another collection object:

Exp:-

```
import java.util.*;

class ArrayDemo2

{
    public static void main(String[] args)

    {
        Collection c=new Vector();

        c.add("nag");

        c.add("arjun");

        ArrayList al=new ArrayList();

        al.add(20);

        al.add(90);

        al.add(80);

        al.add(90);

        al.add(20);

        al.add(90);

        al.add(80);

        al.add(90);

        al.add(c);

        System.out.println("size:"+al.size());

        System.out.println(al);

    }
}
```

Cursors in Java:

To retrieve the Objects one by one from the collection we required a cursor.
there are three cursor support in java

- 1) Enumeration
- 2) Iterator
- 3) ListIterator

Enumeration:

It is a cursor used to read object one by one from collection. Enumeration concept is applicable for only for legacy classes and it is not a universal cursor.

By using Enumeration we can get only read access and we can't perform removable and insertion operations.

We can get Enumeration object by using Element() method.

Exp:-

```
public Enumeration element();
Enumeration e= V.element();
```

Enumeration interface contain the following two methods.

- 1) boolean hasMoreElements()
- 2) object nextElement()

Exp:-

```
import java.util.*;
class VectorDemo
{
    public static void main(String[] args)
    {
        Vector <String>v=new Vector<String>();
```

```
System.out.println("initial capacity:"+v.capacity());  
System.out.println("size:"+v.size());  
v.add("nag");  
v.add("arjun");  
v.add("raj");  
System.out.println(v);  
Enumeration e=v.elements();  
while(e.hasMoreElements())  
{  
    System.out.println(e.nextElement());  
}  
}  
}
```

Iterator:

Iterator interface provide facility iterating elements in forward direction from any collection object.

Iterator cursor is applicable for all the collection hence is called as universal cursor.

To get the Iterator object references we can call the following methods from collection interface.

Syn:- public Iterator iterator();

```
Collection c=new ArrayList();
```

```
Iterator itr=c.iterator();
```

Iterator have the three methods they are

Methods:-

1) boolean hasNext():

it returns true if iterator has more elements.

2) Object next():

It returns the element and moves the cursor point to the next element.

3) void remove():

This method is used to perform removable operations.

Exp:-

```
import java.util.HashSet;  
  
import java.util.Iterator;  
  
class IteratorDemo  
{  
  
    public static void main(String[] args)  
    {  
  
        HashSet hs=new HashSet();  
  
        hs.add("nag");  
  
        hs.add("a");  
  
        hs.add("arjun");  
  
        hs.add("b");  
  
        System.out.println(hs);  
  
        Iterator itr=hs.iterator();  
  
        while(itr.hasNext())  
        {  
  
            String s=(String)itr.next();
```

```
        if(s.equals("a"))

    {

        itr.remove();

    }

    System.out.println(s);

}

System.out.println(hs);

}

}
```

ListIterator:

ListIterator is a child interface of Iterator interface, it is used to retrieve the elements in back and forward direction. hence this cursor is called as a bi-directional cursor.

By using ListIterator it is possible iterate the elements only from list interface implemented classes.

By using ListIterator we can perform retrievable , removable and insertion operation.

To get ListIterator interface object reference the following method from list interface.

Syn:-

```
public ListIterator listIterator();
```

```
List l=new ArrayList();
```

```
ListIterator litr=l.listIterator();
```

The following methods are there in ListIterator Interface

- 1) boolean hasNext()

- 2) Object next()
- 3) boolean hasPrevious()
- 4) Object previous()
- 5) int nextIndex()
- 6) int previousIndex()
- 7) void remove()
- 8) void set(Object)
- 9) void add(Object)

Exp:-

```
import java.util.LinkedList;  
import java.util.ListIterator;  
  
class ListIteratorDemo  
{  
    public static void main(String[] args)  
    {  
        LinkedList ll=new LinkedList();  
  
        //ll.add(10);  
        ll.add("nag");  
        //ll.add(50);  
        ll.add("raj");  
        ll.add("arjun");  
  
        System.out.println(ll);  
  
        ListIterator litr=ll.listIterator();
```

```
System.out.println("data in forward direction");

while(litr.hasNext())
{
    String s=(String)litr.next();

    System.out.println(s);

}

System.out.println(lI);

System.out.println("data in backword direction");

while(litr.hasPrevious())
{
    String s1=(String)litr.previous();

    System.out.println(s1);

}

lI.add("siva");

System.out.println(lI);

}

}
```

Difference between the Enumeration, Iterator and ListIterator:

Enumeration:

This concept is applicable for legacy classes.

By using this cursor we can only read access.

It is single direction cursor. Use only forward direction.

It is introduced in Java 1.0 version.

Iterator:

This is applicable for all collections.

By using this cursor we can perform read access and remove applications.

It is single direction cursor, uses only in forward direction.

introduced in Java 1.2 version.

ListIterator:

It is applicable for ListIterator implementation classes.

By using his cursor we can get read access and also perform insertion and remove operations.

It is bi-directional cursor, retrieve the data in both forward and back word direction.

Introduced in 1.4 version and it is child interface of Iterator.

LinkedList():

LinkedList is an implemented class of List Interface.

LinkedList object maintain the duplicate elements

LinkedList object maintain the elements in the list Insertion order

LinkedList object also accept the homogenous and heterogeneous elements.

LinkedList object accept null pointer constant any number of times.

Advantage:

LinkedList class implements Serializable, Cloneable interface but not random access interface.

LinkedList is best suitable for our frequent operation is insertion and deletion in the middle.

Disadvantage:

If our frequent operation is retrieval operation linked list is not suitable.

Constructors:

1) LinkedList():

this creates an empty LinkedList

Exp:-

```
LinkedList ll=new LinkedList();
```

2) LinkedList(Collection c):

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Exp:-

```
Collection c=new ArrayList();
```

```
LinkedList ll=new LinkedList(c);
```

Methods in LinkedList:

1) void addFirst(Object o):

Insert the specified element at the beginning of this list.

2) void addList(Object o):

Append the specified element to the end of this list.

3) Object getFirst():

Returns the first element in this list.

4) Object getLast():

Returns the last element in this list.

5) Object removeFirst():

Removes and returns the first element from this list.

6) Object removeLast():

Removes and returns the last element from this list.

Exp:-

```
import java.util.LinkedList;
import java.util.ListIterator;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList ll=new LinkedList();
        ll.add(10);
        ll.add(20);
        ll.add(30);
        ll.add(40);
        ll.addFirst(50);
        ll.addLast(60);
        //ll.add('a');
        //ll.removeFirst(60);
        //ll.removeLast("siva");
        System.out.println(ll);
        System.out.println("getfrist="+ll.getFirst());
        System.out.println(ll);
        System.out.println("getLast="+ll.getLast());
        System.out.println("RemoveFirst="+ll.removeFirst());
        System.out.println("RemoveLast="+ll.removeLast());
        System.out.println(ll);
```

}

}

ArrayList:

There is initial capacity

Best suitable for frequent operation is retrieval operation

Not best suitable for frequent operation is insertion and deletion in the middle.

Implementing the Random Access

Direct List Implementation

Maintain duplicates

Null can be added multiple times

Insertion order is maintained

It is Resizable

Implements Serializable

Implements Cloneable

LinkedList:

There is no initial capacity

Not best suitable for frequent operation is retrieval operation

Best suitable for frequent operation is insertion and deletion in middle

Do not implement Random Access

Implements both List and DQueue

Maintain Duplicates

Null can be added multiple times

Insertion order is maintained

It is Resizable

Implements Serializable

Implements Cloneable

Set Interface:

set is an child interface of collection interface.

If you want to represent a group of individual objects as a single entity where duplicate objects are not allowed then we should go for set interface.

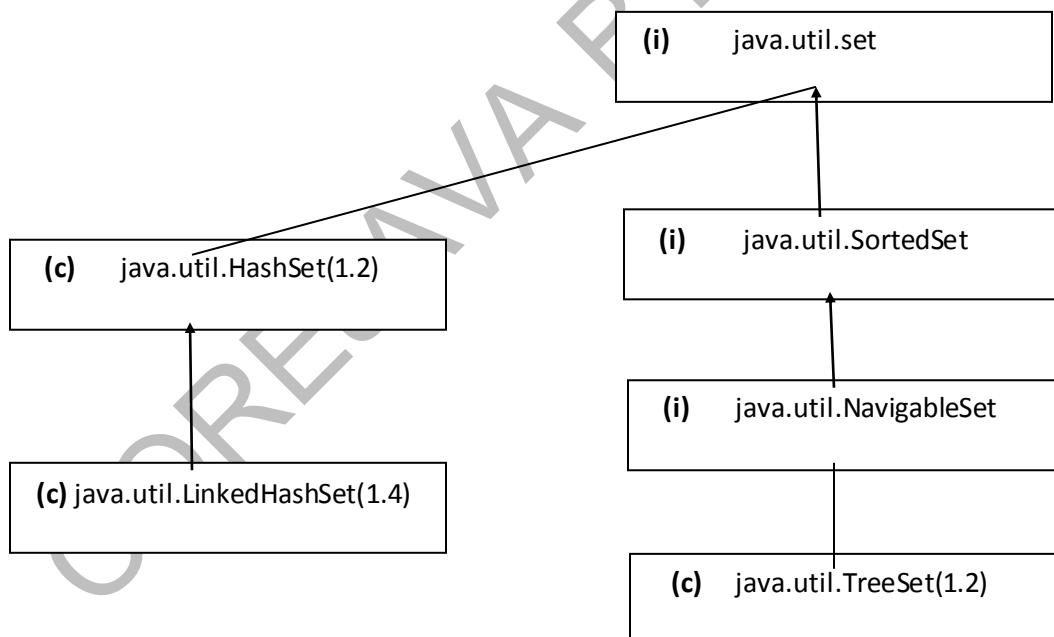
Set Interface does not contain any specific methods, so we can use all the methods which are available in the collection interface.

- **HashSet**

- **LinkedHashSet**

- **TreeSet**

Set interface and its implementation methods:



HashSet:

HashSet is an implemented class of set interface.

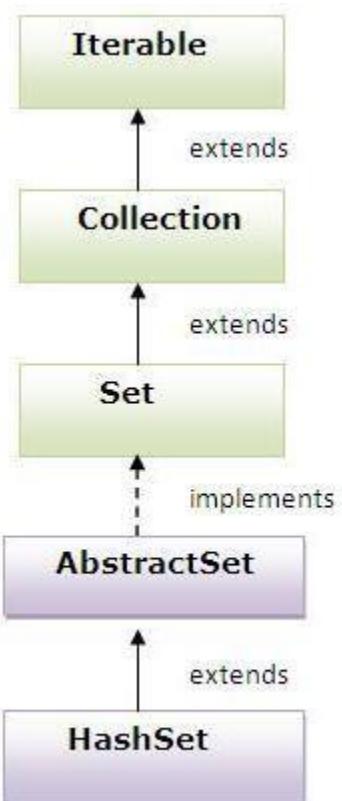
Duplicate objects are not allowed in HashSet by mistake if you are trying to add any duplicate objects, there is no compile time error and no run time exception, simply add method return false.

Insertion order is not followed by HashSet

All the objects are inserted according to hashCode() of the objects

Homogeneous and heterogeneous objects are allowed

Null insertion is possible only one.



Constructors:

HashSet():

```
HashSet hs=new HashSet()
```

This constructor creates an empty HashSet objects with the default initial capacity 16. And default fill ratio is "0.75%", fill ratio is also called as load factor.

HashSet(int initialCapacity):

```
HashSet hs=new HashSet(6);
```

This creates a new HashSet initial capacity and default load factor (0.75)

exp:-

```
HashSet set=new HashSet(6);
```

load factor is 0.75

$$6 * 0.75 = 4.5$$

to above set we can add 4 elements, after adding the 4 elements to the above set if we are adding 5th element then capacity automatically increases to 12 elements.

HashSet(int initialCapacity, float loadFactor):

```
HashSet hs=new HashSet(6,0.5f)
```

It creates a new empty HashSet with the specified initial capacity and with the specified load factor.

To above set after adding the 3 elements if we are adding 4th element to the set then automatically capacity increases to 12 elements.

HashSet(Collection c):

Creates a new HashSet by copying the elements in the specified collection.

Exp:-

```
ArrayList al=new ArrayList();
```

```
al.add(39);
```

```
al.add(45);
```

```
al.add(80);
```

```
HashSet hs=new HashSet(al);
```

Exp:-

```
import java.util.HashSet;

class HashSetDemo

{
    public static void main(String[] args)
    {
        HashSet hs=new HashSet();
        hs.add("nag");
        hs.add("nag");
        hs.add("arjun");
        hs.add("mahesh");
        hs.add("srinu");
        hs.add("ram");
        hs.add("prasad");
        hs.add(10);
        hs.add(60);
        hs.add(null);
        hs.add(null);
        System.out.println(hs);
    }
}
```

LinkedHashSet:

It is a child class of HashSet class

LinkedList is exactly similar to HashSet except following difference.

HashSet:

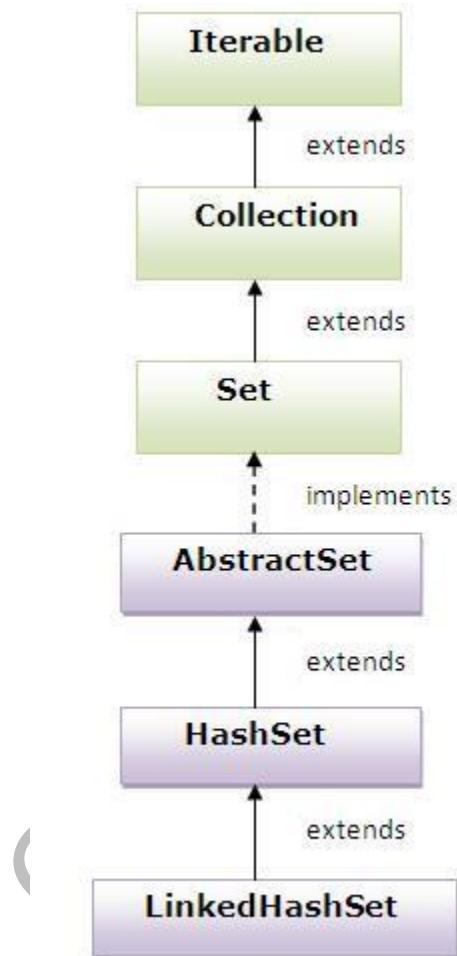
HashSet is not followed insertion order all the objects are inserted on the basics of hashCode.

It is introduced in Java 1.0 version

LinkedHashSet:

LinkedHashSet is follow the insertion order

It is introduced in Java 4.0 version



Exp:-

```
import java.util.LinkedHashSet;  
  
class LinkedHashDemoint  
{  
  
    public static void main(String[] args)  
    {  
  
        LinkedHashSet lsh=new LinkedHashSet();  
  
        lsh.add(20);  
  
        lsh.add(90);  
  
        lsh.add(80);  
  
        lsh.add(90);  
  
        lsh.add(20);  
  
        System.out.println("size:"+lsh.size());  
  
        System.out.println(lsh);  
    }  
}
```

Hashcode:

Syn:- **public int native hashCode()**

hashCode() method is used for retrieving hashcode of the current instance

hashCode() is an identity of an instance using which JVM differentiate different instance of an object

So every instance has unique hashcode generated automatically by JVM converting it's reference into a integer number.

We can get the hashcode in integer number format by calling hashCode() method

We can also override the hashCode method in user defined classes. to create our own hash codes by using object state.

JVM will always uses the hashCode to save the object into Hash Table, HashSet, LinkedHashSet, Hash Map, LinkedHashMap.

Exp:-

```
class Test
{
    public static void main(String[] args)
    {
        A a1=new A();
        A a2=new A();
        System.out.println(a1.hashCode());
        System.out.println(a2.hashCode());
    }
}
```

The above hashCode may change when we run this program in new JVM, but in the same JVM we will get the same hashCode that is creates for first time.

To treat differentiate instance of an object are same, first of all there hashCode should be same

then there state should same.

To generate same hashCode for two instances of an object we must generate hashCode by using
there state not by using reference.

So we must override hashCode() code method in sub class to generate hashCode of its instances

with state.

The logic we are writing inside the hashCode method is called hashing algorithm.

This algorithm may be simple are complex depends on project requirement.

Exp:-

```
class A

{
    int a=10;

    int b=20;

    public boolean equals(Object obj)
    {
        if(obj instanceof A)
        {
            A a1=(A)obj;
            return this.a==a1.a && this.b==a1.b;
        }
        return false;
    }

    public int hashCode()
    {
        return a+b;
    }
}

class TestDemo

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

```
{  
    A a1=new A();  
    A a2=new A();  
    System.out.println(a1.hashCode());//30  
    System.out.println(a2.hashCode());//30  
    System.out.println(a1.equals(a2));//true  
}  
}
```

From the above code we can conclude below two points

- 1) If hashCode of two objects are different then they are different unique objects, we no need to call equals method to confirm it.
- 2) If hashCode of two objects are same, they may be same (or) different unique (or) duplicate objects. Then we must call equals() method to confirm the object are same are different.

Equals:

equals method is used to compare the two objects. In java we can compare object in two ways.

- 3) by using object reference
- 4) by using object state

If two objects of a class are said to be equal only if their reference are state should be equal.

We have to compare the objects either by using “==” operator it always compares the objects with their reference.

By using the equals method we can compare the object based on its implementation.

Object class equals method by default compares the object reference. By overriding the equals method in the user defined class we can compare the object with their state.

Object class equals method implemented as follows.

Exp:-

```
class Object  
{  
    public boolean equals(Object obj)  
    {  
        return (this==obj)  
    }  
}
```

This method returns true if the current object reference is equals to argument object reference otherwise it returns false.

Without Overriding the equals method it will compare the by default reference.

Exp:-

```
class A  
{  
    public static void main(String[] args)  
    {  
        A a1=new A();  
        A a2=new A();  
        //System.out.println(a1.hashCode());  
        //System.out.println(a2.hashCode());  
        System.out.println(a1.equals(a2));  
    }  
}
```

Output:-

1340465859

2106235183

false

Overriding the equals() method it will compare the object state.

Exp:-

```
class A
{
    int a=10;
    int b=20;

    public boolean equals(Object obj)
    {
        if(obj instanceof A)
        {
            A a1=(A)obj;
            return this.a==a1.a && this.b==a1.b;
        }
        return false;
    }

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

```
}

class TestDemo

{

public static void main(String[] args)

{

A a1=new A();

A a2=new A();

System.out.println(a1.hashCode());//30

System.out.println(a2.hashCode());//30

System.out.println(a1.equals(a2));//true

}

}
```

Why we have return if condition:

To avoid the CCE and NPE for retrieving false when heterogeneous objects (or) null is passes, to tell it is unique objects.

Contract between hashCode and equals:

If we are overriding the equals() method then it is always recommended to override the hashCode method because if the equals method returns true by comparing two objects then hashCode of both objects must be same. If equals method return false the hashCode of both objects may or may not be same.

Exp:-

```
import java.util.HashSet;

class Student implements java.io.Serializable

{

private int id ;
```

```
private String name;

public Student(int id , String name)

{

    this.id = id;

    this.name = name;

}

public String toString()

{

    return id+": "+name;

}

public boolean equals(Object obj)

{

//below condition should be presented, else thereis a chance to get

ClassCastException

    if(!(obj instanceof Student))

    {

        {

            return false;

        }

    }

    return this.id==((Student)obj).id;

}

public int hashCode()

{
```

```
        return id;  
    }  
}  
  
class StudentInfo  
{  
  
    public static void main(String[] args)  
{  
  
        Student s1=new Student(1,"nag");  
  
        Student s2=new Student(2,"arjun");  
  
        Student s3=new Student(1,"nag");  
  
        Student s4=new Student(1,"arjun");  
  
        Student s5=new Student(2,"nag");  
  
  
        HashSet hs=new HashSet();  
  
        hs.add(s1);  
  
        hs.add(s2);  
  
        hs.add(s3);  
  
        hs.add(s4);  
  
        hs.add(s5);  
  
        System.out.println(hs);  
    }  
}
```

Output:

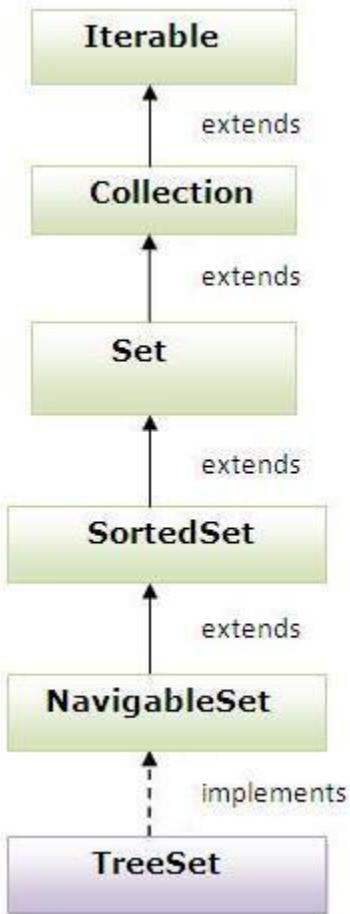
[1: nag, 2: arjun]

By using the hashCode() and equals() methods we can add the unique objects to the Hashtable, HashSet, LinkedHashSet, HashMap, LinkedHashMap. And also we can use those methods for searching and removing elements from the collection.

TreeSet:

TreeSet is implementing NavigableSet interface.

NavigableSet interface is extends SortedSet interface.



Sorted Set Interface(1.2):

SortedSet is child interface of set it is used for represent a group of unique objects (homogeneous) according to some sorting order.

The sorting order either default natural sorting order are customized sorting order.

Sorted set interface defines the following six specific methods.

Object first():

This method returns the first(lowest) element currently in this set.

Object last():

This method returns the last(highest) element currently in this set.

Exp:-

```
import java.util.TreeSet;  
  
class SortedSetDemo  
{  
  
    public static void main(String[] args)  
    {  
  
        TreeSet set=new TreeSet();  
  
        set.add(10);  
  
        set.add(20);  
  
        set.add(30);  
  
        set.add(60);  
  
        set.add(5);  
  
        System.out.println(set);  
  
        System.out.println(set.first());  
  
        System.out.println(set.last());  
    }  
}
```

SortedSet headSet(Object obj):

It returns a view of the portion of this set whose elements are strictly less than toElement.

SortedSet tailSet(Object obj):

It returns a view of the portion of this set whose elements are greater than or equal to object.

Exp:-

```
import java.util.TreeSet;

class SortedSetDemo1

{
    public static void main(String[] args)

    {
        TreeSet set=new TreeSet();

        set.add(10);

        set.add(20);

        set.add(30);

        set.add(60);

        set.add(5);

        System.out.println(set);

        System.out.println("headset upto 30:"+set.headSet(30));//[5, 10, 20]

        System.out.println("headset upto 60:"+set.headSet(60));//[5, 10, 20, 30]

        System.out.println("tailset upto 30:"+set.tailSet(30));//[30,60]

        System.out.println("tailset upto 10:"+set.tailSet(10));//[10,20,30,60]
    }
}
```

SortedSet subSet(Object obj1, Object obj2):

Returns a view of the portion of this set whose elements range is greater than or equal to object1 and less than object2.

Exp:-

```
import java.util.TreeSet;

class SortedSetDemo2

{
    public static void main(String[] args)

    {
        TreeSet set=new TreeSet();

        set.add(10);

        set.add(20);

        set.add(30);

        set.add(60);

        set.add(5);

        System.out.println(set);

        System.out.println("subset from 10 to 60:"+set.subSet(10,60));//[10,20,30,60]

        System.out.println("subset from 5,30:"+set.subSet(5,30));//[5,10,20]

    }
}
```

Comparator comparator():

This method returns comparator used to order the elements in this set, If we are depending the natural sorting order then this method returns null.

NavigableSet Interface:

This interface is introduced in JDK 1.6, the above methods contains the Navigable interface.

Object floor(Object obj):

This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

Exp:-

```
import java.util.TreeSet;  
  
class FloorDemo  
{  
  
    public static void main(String[] args)  
    {  
  
        TreeSet set=new TreeSet();  
  
        set.add(10);  
  
        set.add(20);  
  
        set.add(30);  
  
        set.add(60);  
  
        set.add(5);  
  
        set.add(40);  
  
        System.out.println(set);  
  
        System.out.println(set.floor(55));// here no equal value for 55 so it returns  
        less than value 40  
  
        System.out.println(set.floor(5));// here equal value is there for 5 so it  
        returns 5
```

```
        System.out.println(set.floor(3));// here no equal value and no less than  
        values is there so it returns null  
    }  
}
```

Object Ceiling(Object obj):

This method returns the least element in this set greater than or equal to the given element, or null if there is no such element.

Exp:-

```
import java.util.TreeSet;  
  
class CeilingDemo  
{  
    public static void main(String[] args)  
    {  
        TreeSet set=new TreeSet();  
        set.add(10);  
        set.add(20);  
        set.add(30);  
        set.add(60);  
        set.add(5);  
        set.add(40);  
        System.out.println(set);  
  
        System.out.println(set.ceiling(55));// here no equal value for 55 so it  
        // returns greater value 60  
  
        System.out.println(set.ceiling(5));// here equal value is there for 5 so it  
        // returns 5
```

```
        System.out.println(set.ceiling(61));// here no equal value and no greater  
        than value is there so it returns null  
    }  
}
```

Object lower(object obj):

Returns the greatest element in this set strictly less than the given element, or null if there no such element.

Exp:-

```
import java.util.TreeSet;  
  
class LowerDemo  
{  
    public static void main(String[] args)  
    {  
        TreeSet set=new TreeSet();  
        set.add(10);  
        set.add(20);  
        set.add(30);  
        set.add(60);  
        set.add(5);  
        set.add(40);  
        System.out.println(set);  
        System.out.println(set.lower(40));// less than value 30  
        System.out.println(set.lower(5));// it returns null here why because no  
        lower value  
        System.out.println(set.lower(61));// it retrns less than value 60
```

```
    }  
}  
}
```

Object higher(Object obj):

This method returns the least element in this set strictly greater than the given element, or null if there is no such element.

Exp:-

```
import java.util.TreeSet;  
  
class HigherDemo  
{  
  
    public static void main(String[] args)  
    {  
  
        TreeSet set=new TreeSet();  
  
        set.add(10);  
  
        set.add(20);  
  
        set.add(30);  
  
        set.add(60);  
  
        set.add(5);  
  
        set.add(40);  
  
        System.out.println(set);  
  
        System.out.println(set.higher(40));// it returns greater value 60(compare  
        to 40 next greater value is 60)  
  
        System.out.println(set.higher(5));// //it returns next greater value 10  
  
        System.out.println(set.higher(61));// it returns null here why because no  
        greater value 60 is th last greater value.  
    }  
}
```

```
}
```

Object pollFirst():

This method retrieves and removes the first(lowest) element, or returns null if this set is empty.

Object pollLast();

This method retrieves and removes the last(highest) element, or returns null if this set is empty.

Exp:-

```
import java.util.TreeSet;

class PollFirstDemo

{
    public static void main(String[] args)
    {
        TreeSet set=new TreeSet();
        TreeSet set1=new TreeSet();
        set.add(10);
        set.add(20);
        set.add(30);
        set.add(60);
        set.add(5);
        set.add(40);
        System.out.println(set); // [5,10,20,30,40,60]
        System.out.println(set.pollFirst()); // 5
```

```
        System.out.println(set); // [10, 20, 30, 40, 60]  
        System.out.println(set.pollFirst()); // [10]  
        System.out.println(set); // [20, 30, 40, 60]  
        System.out.println(set.pollLast()); // [60]  
        System.out.println(set1.pollFirst()); // this returns null why because it is  
        empty set  
        System.out.println(set1.pollLast()); // this returns null why because it is  
        empty set  
    }  
}
```

NavigableSet descendingSet():

This methods returns a reverse order view of the elements contained in this set.

Exp:-

```
import java.util.TreeSet;  
  
class DescendingDemo  
{  
    public static void main(String[] args)  
    {  
        TreeSet set=new TreeSet();  
        TreeSet set1=new TreeSet();  
        set.add(10);  
        set.add(20);  
        set.add(30);  
        set.add(60);
```

```
        set.add(5);

        set.add(40);

        System.out.println(set);//[5,10,20,30,40,60]

        System.out.println(set.descendingSet());//[60,40,30,20,10,5]

    }

}
```

TreeSet:

The implemented data structure of TreeSet is Red-Black Tree backed by Tree Map Instance

TreeSet is not allowed the Duplicate elements, if we are trying to insert duplicate then add() method return false

TreeSet allows the homogeneous and unique elements.

TreeSet is not allowed the null it leads NPE(null pointer Exception)

TreeSet is not followed insertion order, all the objects are inserted based on some sorting order. The sorting order may be either default natural sorting order (or) any customized sorting order described by comparator object.

If we are depending on any default natural sorting order then that objects must be homogeneous and comparable

Note: The object is called as comparable object then that class should be implement comparable interface. All the Wrapper classes and String class implemented comparable interface, hence all Wrapper objects and String objects is comparable objects.

If we are defining any customized sorting order TreeSet is allowed non-comparable objects.

Null Acceptances: For the Empty TreeSet at the first element null insertion is possible. But after inserting the null if we are trying to insert any other element we will get Null Pointer Exception(it is up to valid Java 6.0 version).

Exp:-

```
import java.util.TreeSet;  
  
class NullDemo  
  
{  
  
    public static void main(String[] args)  
  
    {  
  
        TreeSet set=new TreeSet();  
  
        set.add(null);  
  
        System.out.println(set);  
  
    }  
  
}
```

Output:

Null

Exp:-

```
import java.util.TreeSet;  
  
class NullDemo  
  
{  
  
    public static void main(String[] args)  
  
    {  
  
        TreeSet set=new TreeSet();  
  
        set.add(null);  
  
        set.add(10);  
  
        System.out.println(set);  
  
    }  
  
}
```

```
}
```

Output:

Error: java.lang.NullPointerException

From Java 7.0 version on words an empty TreeSet is also not allowed null value.

For the non-empty TreeSet if we are trying to insert null we will get Null Pointer Exception.

Exp:-

```
TreeSet set= new TreeSet();  
  
ts.add(null);  
  
// valid upto java 6.0 version, from java 7.0 version on words null is not allowed.
```

Exp:-

```
import java.util.TreeSet;  
  
class TreeDemolnt  
{  
  
    public static void main(String[] args)  
    {  
  
        TreeSet ts=new TreeSet();  
  
        ts.add(20);  
  
        ts.add(90);  
  
        ts.add(80);  
  
        ts.add(90);  
  
        ts.add(20);  
  
        //ts.add("nag"); //classcastException  
  
        //ts.add(null); // NullPointerException
```

```

        System.out.println("size:"+ts.size());

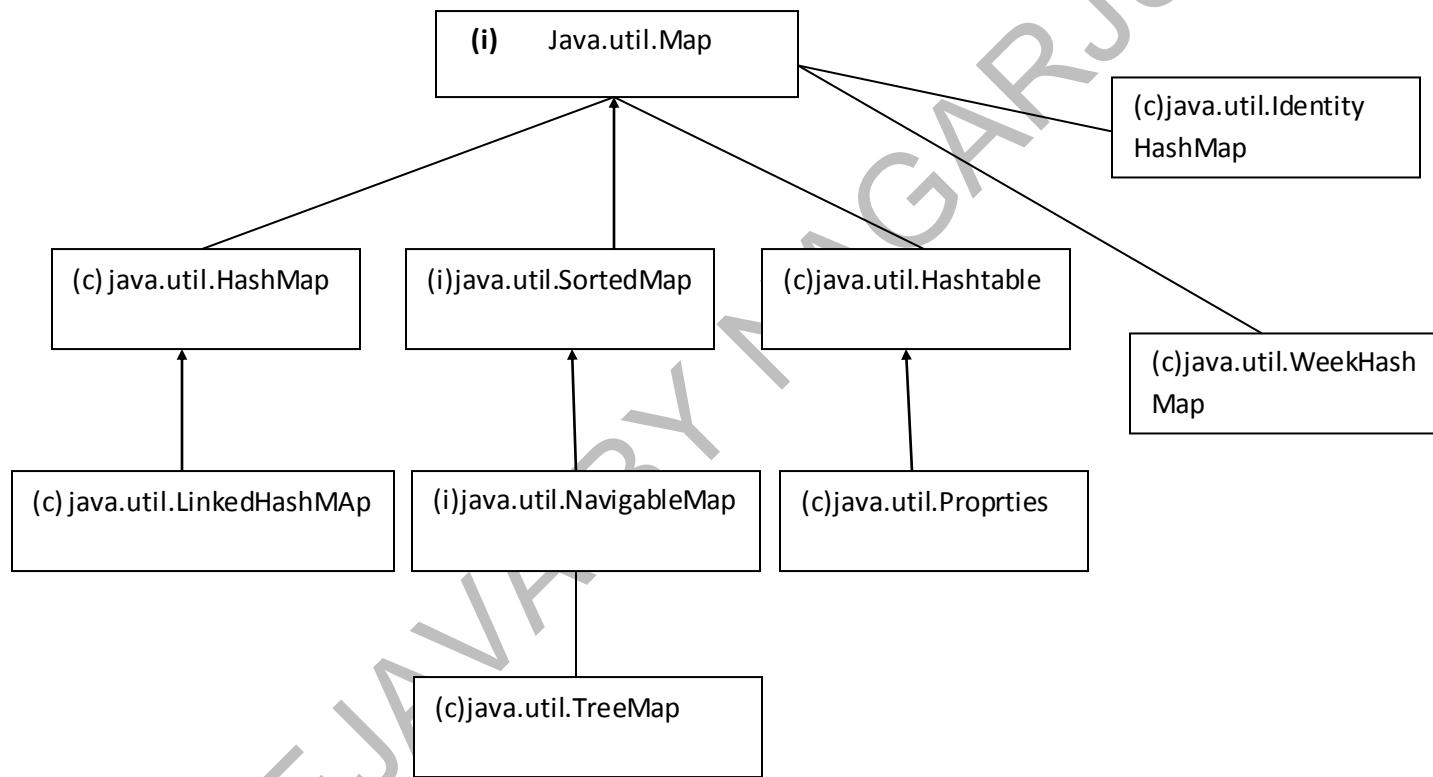
        System.out.println(ts);

    }

}

```

Map Interface and it's implementations:



Map is not a child class interface of collection, it is also act as root interface in collections.

If you want to represent a group of objects as a key and value pairs then we should go for Map.

Here both key and values are objects.

Map object

key	value
un	nag

Above each key and value pair is called as one entry.

Duplicate keys are not allowed but value can be duplicated.

Methods:

Object put(object key, object value):

put method can be used to insert one entry into the map. if the specified key already available than the old value can be replaced with the new value and new value will be returned.

Exp:-

```
import java.util.*;  
  
class HashtableDemo1  
{  
  
    public static void main(String args[])  
    {  
  
        Hashtable hm=new Hashtable();  
  
        hm.put(100,"Amit");  
  
        hm.put(102,"Ravi");  
  
        hm.put(100,"nag");  
  
        System.out.println(hm);  
  
    }  
  
}
```

public void putAll(Map map):

It is used to insert the specified map in this map.

public Object remove(object key):

This method removes the entry associated with the specified key and return corresponding value, if the specified key is not available means it returns null.

public Object get(Object key):

It returns the value associated with key, if the specified key is not available then get() method returns null.

public boolean containsKey(Object key):

It is used to search the specified key from this map.

public boolean containsValue(Object value):

It is used to search the specified value from this map.

public Set keySet():

It returns the Set view containing all the keys.

public Set entrySet():

This method returns the Set view containing all the keys and values.

Public boolean isEmpty():

public int size():

public void clear():

Collection values():

It returns collection with map object values.

Hashtable:

A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashCode() method. A Hashtable contains values based on the key. It implements the Map interface and extends Dictionary class.

It contains only unique elements.

Key: it is not allowed duplicate key, **Value:** it is allowed the duplicate values

Can't store elements in sorted order

It is synchronized

It may have not have any null key or value.

Constructors:

Hashtable():

Construct an empty Hashtable with the default initial capacity(11) and the default load factor (0.75).

Hashtable(int initialCapacity):

Constructs an empty Hashtable with the specified initial capacity and the default load factor(0.75)

Hashtable(int initialCapacity, float loadFactor):

Construct an empty Hashtable with the specified initial capacity and load factor

Hashtable(map):

Construct a new Hashtable with the same mappings as the specified Map.

Exp:-

```
import java.util.*;
class HashtableDemo
{
    public static void main(String args[])
    {
        Hashtable hm=new Hashtable();
        hm.put(100,"Amit");
    }
}
```

```
hm.put(102,"Ravi");

hm.put(101,"Vijay");

hm.put(103,"Rahul");

hm.put(101,"arjun");// here vijay replace the arjun

hm.put(106,"arjun");

hm.put(null,"null");// NullPointerException

System.out.println(hm);

}

}
```

Retrieving the key and values from the Hashtable.

Exp:-

```
import java.util.*;

class HashtableDemo1

{

public static void main(String args[])

{

Hashtable hm=new Hashtable();

hm.put(100,"Amit");

hm.put(102,"Ravi");

hm.put(101,"Vijay");

hm.put(103,"Rahul");

hm.put(101,"arjun");

hm.put(106,"arjun");

//hm.put(null,"null");// NullPointerException
```

```
System.out.println(hm);

Set set=hm.entrySet();

Iterator itr=set.iterator();

while(itr.hasNext()){

Map.Entry m=(Map.Entry)itr.next();

System.out.println(m.getKey()+" "+m.getValue());

}

}

}
```

Output:

{106=arjun, 103=Rahul, 102=Ravi, 101=arjun, 100=Amit}

106 arjun

103 Rahul

102 Ravi

101 arjun

100 Amit

HashMap:

HashMap implemented data structure is Hashtable.

HashMap is a implemented class of Map interface

HashMap can be used to represent a group of objects as key and value pairs

Homogenous and heterogeneous unique objects are allowed for key

Homogenous and heterogeneous unique and duplicate objects allowed for value

Insertion order is not followed because insertion is based on hashCode of keys

Null is allowed only one for key, but null is allowed more than one for value

It is introduced in java 1.2 version.

Constructors:

HashMap():

Construct an empty HashMap with the default initial capacity(16) and the default load factor (0.75).

HashMap(int initialCapacity):

Constructs an empty HashMap with the specified initial capacity and the default load factor(0.75)

HashMap(int initialCapacity, float loadFactor):

Construct an empty HashMap with the specified initial capacity and load factor

HashMap(map):

Construct a new HashMap with the same mappings as the specified Map.

Exp:-

```
import java.util.HashMap;  
  
class HashMapDemo2  
{  
  
    public static void main(String[] args)  
    {  
  
        HashMap hm=new HashMap();  
  
        hm.put(1,"nag");  
  
        hm.put(2,"arjun");  
  
        hm.put(3,"raj");  
  
        hm.put(2,"nag1");  
  
        hm.put(null,"null");
```

```
        System.out.println(hm);
    }
}
```

Retrieving the key and values from HashMap

Exp:-

```
import java.util.*;
class HashMap2
{
    public static void main(String args[])
    {
```

```
    HashMap hm=new HashMap();
```

```
    hm.put(100,"Amit");
    hm.put(102,"Ravi");
    hm.put(100,"nag");
    hm.put(null,null);
    hm.put(100,null);
    System.out.println(hm);
    Set set=hm.entrySet();
    Iterator itr=set.iterator();
    while(itr.hasNext())
    {
```

```
        Map.Entry m=(Map.Entry)itr.next();
```

```
System.out.println(m.getKey()+" "+m.getValue());  
}  
}  
}
```

Difference between the Hashtable and HashMap:

Hashtable	HashMap
1) All methods are synchronized	1) No method is synchronized
2) Hashtable is thread safe	2) HashMap object is not a thread safe
3) Performance id good	3) performance is poor
4) Null is allowed both key and value	4) Null is not allowed both key and value
5) Introduced in 1. 2 version it is legacy class.	5) Introduced in 1.0 version it is legacy class.

LinkedHashMap:

It is exactly same as HashMap except the following difference.

HashMap	LinkedHashMap
1) Underlying data structure is Hashtable	1) Underlying data structure is Hashtable + LinkedList
2) Insertion is not maintained	2) Insertion order is maintained
3) Introduced in java 1.2 version	3) Introduced in java 1.4 version

Exp:-

```
import java.util.*;  
  
class LinkedHashMap2  
{  
  
public static void main(String args[])
}
```

```
{  
LinkedHashMap lhm=new LinkedHashMap();  
lhm.put(100,"Amit");  
lhm.put(102,"Ravi");  
lhm.put(100,"nag");  
lhm.put(null,null);  
lhm.put(100,null);  
System.out.println(lhm);  
Set set=lhm.entrySet();  
Iterator itr=set.iterator();  
while(itr.hasNext())  
{  
Map.Entry m=(Map.Entry)itr.next();  
System.out.println(m.getKey()+" "+m.getValue());  
}  
}  
}
```

IdentityHashMap:

IdentityHashMap is an implemented class of Map interface.

It is exactly same as HashMap except following difference

In the case of HashMap to identify duplicate keys JVM always uses equals() method which is mostly meant for content comparison.

But in the case of IdentityHashMap JVM uses "==" operator to identify duplicate keys which is always meant for reference comparison.

Exp:-

```
import java.util.*;  
  
class IdentityHashMapDemo  
{  
  
    public static void main(String args[])  
    {  
  
        HashMap hm=new HashMap();  
  
        IdentityHashMap ihm=new IdentityHashMap();  
  
        Integer i1=new Integer(10);  
  
        Integer i2=new Integer(10);  
  
        hm.put(i1,"nag");  
  
        hm.put(i2,"arjun");  
  
        System.out.println(hm);  
  
        ihm.put(i1,"nag");  
  
        ihm.put(i2,"arjun");  
  
        System.out.println(ihm);  
  
        System.out.println(i1==i2);  
  
        System.out.println(i1.equals(i2));  
    }  
}
```

Output:

```
{10=arjun}  
  
{10=arjun, 10=nag}  
  
false
```

true

The above HashMap i1,i2 are duplicate keys according to "equals()" method, hence old value replace with the new value.

In the case of IdentityHashMap i1,i2 are not duplicate keys because it uses "i1==i2" its compares the reference means "i1==i" return false.

WeakHashMap:

It is exactly same as HashMap except the following difference

In the case of HashMap an object is not eligible for garbage collection if it is associated with HashMap even though it doesn't have any external reference

Note: HashMap dominates garbage Collector.

In the case of WeakHashMap an object is eligible for GC if it doesn't contain any external reference even it is associated with WeakHashMap.

Note: Garbage Collector dominates the WeakHashMap

HashMap Example:

```
import java.util.*;  
  
class Student  
{  
    private int id;  
    private String name;  
    public Student(int id, String name)  
    {  
        this.id=id;  
        this.name=name;  
    }
```

```
public void finalize()
{
    System.out.println("id="+id+",name="+name+",student obj is going to be deleted");
}

public String toString()
{
    return "("+id+","+name+")";
}

}

class GCWithHMDemo
{

public static void main(String[] args)throws Exception
{
    Student s1,s2;
    s1=new Student(39,"nag");
    s2=new Student(38,"arjun");
    HashMap hm=new HashMap();
    hm.put(s1,"java");
    hm.put(s2,"advjava");
    s2=null;
    System.out.println(hm);
    System.gc();
    Thread.sleep(500);
}
```

```
}
```

Output:

```
{(39,nag)=java, (38,arjun)=advjava}
```

WeakHashMap Example:

Exp:-

```
import java.util.*;  
  
class Student  
{  
    private int id;  
    private String name;  
    public Student(int id, String name)  
    {  
        this.id=id;  
        this.name=name;  
    }  
    public void finalize()  
    {  
        System.out.println("id="+id+",name="+name+",student obj is going to be deleted");  
    }  
    public String toString()  
    {  
        return "("+id+","+name+");  
    }  
}
```

```
class GCWithWHMDemo
{
    public static void main(String[] args) throws Exception
    {
        Student s1,s2;
        s1=new Student(39,"nag");
        s2=new Student(38,"arjun");
        WeakHashMap whm=new WeakHashMap();
        whm.put(s1,"java");
        whm.put(s2,"advjava");
        s2=null;
        System.out.println(whm);
        System.gc();
        Thread.sleep(500);
    }
}
```

Output:

```
{(39,nag)=java, (38,arjun)= advjava}
id=38,name=arjun, student obj is going to be deleted
```

SortedMap interface:

If we want to represent a group of entries according to some sorting order of keys then we should go for SortedMap.

TreeMap:

TreeMap is class that implements NavigableMap interface, NavigableMap

is extending SortedMap

The implemented data Structure of TreeMap is RED-BLACK Tree

Homogeneous unique objects comparable for key, Homogeneous and heterogeneous unique and duplicate objects for value

Natural sorting order , randomly by passing key sequentially in the order stored

Key: Null is not allowed the key it leads to NPE

Value: Null is allowed more than one for value.

Constructors:

The following three constructors will arrange TreeMap elements based on natural order of its keys. Means TreeMap keys must be comparable type.

TreeMap():

Constructs a new , empty tree map, using the natural sorting ordering of its keys.

TreeMap(map):

Constructs a new tree map containing the same mappings as the given map, ordered according to the natural ordering of its keys.

Exp:-

```
import java.util.*;
class TreeMapExp
{
    public static void main(String[] args)
    {
        TreeMap tm=new TreeMap();
        tm.put(1,"nag");
    }
}
```

```
tm.put(2,"arjun");

tm.put(3,"raj");

tm.put(2,"nagarjuna");

System.out.println(tm);

}

}
```

Retrieving the elements form TreeMap:

Exp:-

```
import java.util.*;

class TreeMapExp{

public static void main(String args[]){

TreeMap hm=new TreeMap();

hm.put(100,"Amit");

hm.put(102,"Ravi");

hm.put(101,"Vijay");

hm.put(103,"Rahul");

System.out.println(hm);

Set set=hm.entrySet();

Iterator itr=set.iterator();

while(itr.hasNext()){

Map.Entry m=(Map.Entry)itr.next();

System.out.println(m.getKey()+" "+m.getValue());

}

}

}
```

```
}
```

Output:

100 Amit

101 Vijay

102 Ravi

103 Rahul

TreeMap(SortedMap):

Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

Exp:-

```
import java.util.*;  
  
class Faculty implements Comparable<Faculty>  
{  
    private int id;  
    private String name;  
    private int age;  
  
    public Faculty(int id,String name,int age)  
    {  
        this.id=id;  
        this.name=name;  
        this.age=age;  
    }  
  
    public String toString()  
    {
```

```
        return "("+id+","+name+","+age+")";  
    }  
  
    public int compareTo(Faculty f)  
    {  
        return this.id - f.id;  
    }
```

If we want compare the name means use below code

```
/*  
public int compareTo(Faculty f)  
{  
    String a=this.name;  
    String b=f.name;  
    int n=a.compareTo(b);  
    return n;  
}  
*/  
}  
}  
  
class TreeMapDemo  
{  
    public static void main(String[] args)  
    {  
        Faculty f1=new Faculty(1,"nag",25);  
        Faculty f2=new Faculty(3,"arjun",26);  
        Faculty f3=new Faculty(2,"raj",18);  
    }
```

```
TreeMap<Faculty,String> map=new TreeMap<Faculty,String>();  
  
map.put(f1,"java");  
  
map.put(f2,"advjava");  
  
map.put(f3,"MTECH");  
  
System.out.println(map);  
  
}  
  
}
```

TreeMap(Comparator):

This method Construct a new empty tree map, ordered according to the given comparator.

Exp:-

```
import java.util.*;  
  
class Faculty  
  
{  
  
private int id;  
  
private String name;  
  
private int age;  
  
public Faculty(int id,String name,int age)  
  
{  
  
this.id=id;  
  
this.name=name;  
  
this.age=age;  
  
}  
  
public String toString()
```

```
{  
    return "("+id+","+name+","+age+");  
}  
  
public int getId()  
{  
    return id;  
}  
  
public String getName()  
{  
    return name;  
}  
  
public int getAge()  
{  
    return age;  
}  
}  
  
class IdComparator implements Comparator<Faculty>  
{  
    public int compare(Faculty f1,Faculty f2)  
    {  
        return f1.getId()-f2.getId();  
    }  
}  
  
class NameComparator implements Comparator<Faculty>
```

```
{  
public int compare(Faculty f1,Faculty f2)  
{  
String s1=f1.getName();  
String s2=f2.getName();  
int n=s1.compareTo(s2);  
return n;  
}  
}  
  
class AgeComparator implements Comparator<Faculty>  
{  
public int compare(Faculty f1,Faculty f2)  
{  
return f1.getAge()-f2.getAge();  
}  
}  
  
class TreeMapDemoComparator  
{  
public static void main(String[] args)  
{  
IdComparator idc=new IdComparator();  
NameComparator nc=new NameComparator();  
AgeComparator ac=new AgeComparator();
```

```
Faculty f1=new Faculty(1,"nag",25);
Faculty f2=new Faculty(3,"arjun",26);
Faculty f3=new Faculty(2,"raj",18);
TreeMap<Faculty,String> map1=new TreeMap<Faculty,String>(idc);
TreeMap<Faculty,String> map2=new TreeMap<Faculty,String>(nc);
TreeMap<Faculty,String> map3=new TreeMap<Faculty,String>(ac);
map1.put(f1,"java");
map1.put(f2,"advjava");
map1.put(f3,"MTECH");
map2.put(f1,"java");
map2.put(f2,"advjava");
map2.put(f3,"MTECH");
map3.put(f1,"java");
map3.put(f2,"advjava");
map3.put(f3,"MTECH");
System.out.println(map1);
System.out.println(map2);
System.out.println(map3);
}
```

Output:

```
{(1,nag,25)=java, (2,raj,18)=MTECH, (3,arjun,26)=advjava}// compare by id
{(3,arjun,26)=advjava, (1,nag,25)=java, (2,raj,18)=MTECH}// compare by name
```

```
{(2,raj,18)=MTECH, (1,nag,25)=java, (3,arjun,26)=advjava}// compare by age
```

Q) Why compare() method have two parameters and compareTo() method have one parameter?

CompareTo(): compareTo() method is called on currently adding object so the two objects adding object and ts element are passed into compareTo() method as CO(current object) and AO(adding object). Hence it has only one parameter.

Compare(): compare() method is not allowed on currently adding object, so the two objects adding object and ts element must be passed in compare() method as a argument. Hence it has two parameters.

Q) When a class should override hashCode() , equals(), compareTo() and compare methods?

Add the objects to all the set and map implemented collections we must override the above methods.

- 1) We must override hashCode() and equals() methods to add the instance to HS, LSH, HM, LHM, HT. If we do not override there is no Compile time error and no Run time error, the problem is objects are not found with new objects.
- 2) We must override compareTo() method to add the instances to TS, TM collection objects, if we are not override there is no Compile time error but it leads to Run time exception(class caste exception). If we still want to add the instances we must add them with custom comparator object. In this case we must override compare() method in another class.

All the above four methods we must implement in the sub class to add the instances of all collection classes.

Q) List implemented classes are uses hashCode() and equals() methods?

They use only equals() method only in searching and removing operation not while adding.

Working with TreeSet and TreeMap classes:

TreeSet and TreeMap classes are used for sorting objects and entries in sorting order

according to the objects by key natural sorting order.

The natural sorting order means the sorting order that is provided in this objects class in most of the object case the natural sorting order is ascending.

For String and Wrapper classes the natural sorting order is ascending order. It means when we add String and Wrapper classes objects to TreeSet.

- String objects are sorting in alphabetical ascending order number.
- Wrapper classes objects are stored in number ascending order.
- Character Wrapper classes objects are stored in its character ASCII number Ascending order.
- Boolean Wrapper classes objects are stored in alphabetical order false, true.

Note: String Buffer and String Builder classes are non-comparable types, That means these two classes are not implementing comparable interface. So we can't add this two objects instances to TreeSet (or) TreeMap directly. But we can add indirectly by using the comparator.

Rule:

- 1) To add instances of class to TreeSet (or) TreeMap classes those objects must be subclasses of java.lang. Comparable interface.
- 2) And should implement its method compareTo(Object o) with object sorting logic for deciding current adding object position among the element added to TreeSet/TreeMap.
- 3) The above method must be called from TreeSet class add() method, and should implements it in every object whose instances want to be added to TreeSet(or) TreeMap.
- 4) Based on the value returned from this compareTo() method TreeSet (or) TreeMap classes add()/put() methods decided the position of the element.

If it is returns

- a) -Ve number, element is added LEFT to current TreeSet element.

- b) +Ve number, element is added RIGHT to current TreeSet element.
 - c) 0 means, element is not added.
- 5) We can also add objects to TreeSet by using java.util.Comparator interface, this interface is used for developing the custom sorting order logic.

Custom Sorting Order means

- a) Reverse natural sorting order of comparable objects
 - (or)
- b) Sorting order logic for adding the non-comparable objects.

Properties class:

The java util package provides many utility interfaces and classes for easy manipulation of in-memory data. Among those classes Properties is one class.

Properties class is a sub class of Hashtable.

Hashtable is a table that maps keys to values, in Hashtable keys can be any type values can be any type.

The **properties** object contains key and value pair both as a string.

By using Properties class object we can store the properties to a . properties(or) .xml file

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

Advantage of properties file

Easy Maintenance: If any information is changed from the properties file, you don't need to recompile the java class. It is mainly used to contain variable information i.e. to be changed.

Methods of Properties class

public void load(Reader r): loads data from the Reader object.

public void load(InputStream is):

loads data from the InputStream object

public String getProperty(String key):

returns value based on the key.

public void setProperty(String key, String value):

sets the property in the properties object.

public void store(Writer w, String comment):

writers the properties in the writer object.

public void store(OutputStream os, String comment):

writes the properties in the OutputStream object.

storeToXML(OutputStream os, String comment):

writers the properties in the writer object for generating xml document.

public void storeToXML(Writer w, String comment, String encoding):

writers the properties in the writer object for generating xml document with specified encoding.

Example of Properties class to create properties file

Exp:-

```
import java.util.*;
import java.io.*;
public class ProperiesCreate {
    public static void main(String[] args) throws Exception{
        Properties p = new Properties();
        p.setProperty("name", "nag");
        p.setProperty("email", "nagarjunajava3@gmail.com");
```

```
p.store(new FileWriter("E:/nag.properties")," Properties File Data ");

System.out.println("File created successfully");

}

}
```

Properties class to get information from properties file

Exp:-

```
import java.util.*;

import java.io.*;

public class PropertiesTest

{

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

    {

        FileReader reader=new FileReader("E:/nag.properties");

        Properties p=new Properties();

        p.load(reader);

        System.out.println(p.getProperty("name"));

        System.out.println(p.getProperty("email"));

    }

}
```

Properties class to get all the system properties

By `System.getProperties()` method we can get all the properties of system. Let's create the class that gets information from the system properties.

Exp:-

```
import java.util.*;
import java.io.*;
public class SystemProperties {
    public static void main(String[] args) throws Exception{
        Properties p=System.getProperties();
        Set set=p.entrySet();
        Iterator itr=set.iterator();
        while(itr.hasNext()){
            Map.Entry entry=(Map.Entry)itr.next();
            System.out.println(entry.getKey()+" = "+entry.getValue());
        }
    }
}
```

Exp:-

```
import java.util.Properties;
class ProDemo
{
    public static void main(String[] args)
    {
        Properties p=new Properties();
        p.setProperty("dbuser","System");
    }
}
```

```
p.setProperty("dbpassword","System");

String s1=p.getProperty("dbuser");

String s2=p.getProperty("dbpassword");

String s3=p.getProperty("dbname");

String s4=p.getProperty("dbname","oracle");

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

System.out.println(s4);

}

}
```

Output:

System

System

null

oracle

Exp:-

```
import java.util.Properties;

class ProDemo1

{

public static void main(String[] args)

{

Properties p=new Properties();
```

```
p.setProperty("dbuser","System");

p.setProperty("dbpassword","nag");

Properties p1=new Properties(p);

System.out.println(p.getProperty("dbuser"));

System.out.println(p1.getProperty("dbuser"));

System.out.println(p1.getProperty("dbpassword"));

//p1.list(System.out);

//p.list(System.out);

}

}
```

Output:

```
System

System

nag

-- listing properties --

dbpassword=nag

dbuser=System

-- listing properties --

dbpassword=nag

dbuser=System
```

Exp:-

```
import java.util.Properties;

class ProDemo2

{
```

```
public static void main(String[] args)
{
    Properties p=new Properties();
    p.setProperty("dbuser","System");
    p.setProperty("dbpassword","nag");
    p.list(System.out);
}
}
```

Output:

```
-- listing properties --
dbpassword=nag
dbuser=System
```

Ex:- Write a program to create .properties file using the PrintStream

```
import java.util.Properties;
import java.io.*;
class ProDemoPrintStream
{
    public static void main(String[] args) throws Exception
    {
        PrintStream ps=new PrintStream("E:/nag1.properties");
        Properties p=new Properties();
        p.setProperty("dbuser","System");
        p.setProperty("dbpassword","nag");
        p.list(ps);
    }
}
```

```
        System.out.println("the .properties file is created successfully");

    }

}
```

Output:

the .properties file is created successfully

EEx:- Write a program to create .xml file using the PrintStream

```
import java.util.Properties;

import java.io.*;

class ProDemoPrintStream

{

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

    {

        PrintStream ps=new PrintStream("E:/nag2.xml");

        Properties p=new Properties();

        p.setProperty("dbuser","System");

        p.setProperty("dbpassword","nag");

        p.list(ps);

        System.out.println("the .xml file is created successfully");

    }

}
```

Output:

the .xml file is created successfully

Ex:- Write a program to create .xml file using the FileOutputStream

```
import java.util.*;
import java.io.*;
public class PropertiesCreate {
    public static void main(String[] args) throws Exception{
        Properties p=new Properties();
        p.setProperty("name","nag");
        p.setProperty("email","nagarunajava3@gmail.com");
        FileOutputStream fos=new FileOutputStream("E:/student1.xml");
        p.storeToXML(fos,"userinfo");
        System.out.println("File created successfully");
    }
}
```

Ex:- Write a program to create .properties file using the FileOutputStream

```
import java.util.*;
import java.io.*;
public class PropertiesCreate {
    public static void main(String[] args) throws Exception{
        Properties p=new Properties();
        p.setProperty("name","nag");
        p.setProperty("email","nagarunajava3@gmail.com");
        FileOutputStream fos=new FileOutputStream("E:/student1.properties");
        p.store(fos,"userinfo");
        System.out.println("File created successfully");
    }
}
```

```
}
```

```
}
```

Ex:- Write a program to load .properties file using the FileInputStream

```
import java.util.Properties;  
  
import java.io.*;  
  
class ProDemoFileInputStream  
{  
  
    public static void main(String[] args) throws Exception  
{  
  
        Properties p = new Properties();  
  
        System.out.println("before loading");  
  
        System.out.println(p.getProperty("dbuser"));  
  
        FileInputStream fis = new FileInputStream("E:/nag1.properties");  
  
        p.load(fis);  
  
        System.out.println("after loading");  
  
        System.out.println(p.getProperty("dbuser"));  
    }  
}
```

Output:

before loading

null

after loading

System

Ex:- Write a program to load .xml file using the FileInputStream

```
import java.util.Properties;  
import java.io.*;  
  
class ProDemoFileInputStreamXml  
{  
  
    public static void main(String[] args) throws Exception  
{  
  
        Properties p=new Properties();  
  
        System.out.println("before loading");  
  
        System.out.println(p.getProperty("email"));  
  
        FileInputStream fis=new FileInputStream("E:/student1.xml");  
  
        p.loadFromXML(fis);  
  
        System.out.println("after loading");  
  
        System.out.println(p.getProperty("email"));  
    }  
}
```

Following table gives the properties of derived classes.

Data Structure	Interface	Duplicates	Methods available	DS that inherits
HashSet	Set	Unique elements	equals(), hashCode()	Hashtable
LinkedHashSet	Set	Unique elements	equals(), hashCode()	Hashtable, doubly-linked list
TreeSet	SortedSet	Unique elements	equals(), compareTo()	Balanced Tree
ArrayList	List	Duplicates allowed	equals()	Resizable array
LinkedList	List	Duplicates allowed	equals()	Linked list
Vector	List	Duplicates allowed	equals()	Resizable array
HashMap	Map	Unique keys	equals() and hashCode()	Hash table
LinkedHashMap	Map	Unique keys	equals() and hashCode()	Hash table and doubly-linked list
Hashtable	Map	Unique keys	equals(), hashCode()	Hash table
TreeMap	SortedMap	Unique keys	equals(), compareTo()	Tree Map

Collection is the root interface for all the hierarchy (except Map).

Set interface unique feature is that it does not accept duplicate elements. That is, no two elements will be the same.

SortedSet interface is derived from Set interface and adds one more feature that the elements are arranged in sorted order by default.

List interface permits duplicate elements.

Queue interface holds elements and returns in FIFO order.

adds the elements in key/value pairs. **Duplicate keys are not allowed**, but duplicate values are allowed. One key can map one value only.

SortedMap interface is a particular case of Map. Keys are sorted by default.

transient keyword

The transient keyword is used in serialization. If you define any data member as transient, it will not be serialized. Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

Example of transient keyword

In this example, we have created the two classes Student and Persist. One data member of the Student class is declared as transient, its value will not be serialized. If you de-serialize the object, it will return the default value for transient variable.

Exp:-

```
import java.io.*;

public class SerializeStudent implements Serializable
{
    int id;
    String name;

    public SerializeStudent(int id, String name)
    {
        this.id = id;
        this.name = name;
        transient int age; //Now it will not be serialized
```

```
}

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

{

SerializeStudent s1 =new SerializeStudent(101,"nag",26);

FileOutputStream fout=new FileOutputStream("D:/n.txt");

ObjectOutputStream out=new ObjectOutputStream(fout);

out.writeObject(s1);

out.flush();

System.out.println("success");

}

}


```

Output: Success