



An ISO 9001:2008 Certified Company

**n|u|m|e|r|i|c**  
infosystem private limited

# JAVA PROGRAMMING for the absolute beginner



"FIRST SOLVE THE PROBLEM.  
THEN, WRITE THE CODE "



6

Weeks / Months

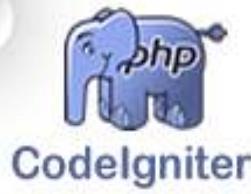
Project Based  
Industrial Training



Training With Real Time Project



Java



python™



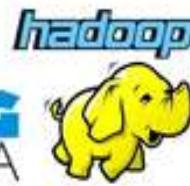
ANDROID



Embedded C  
&  
Robotics



MATLAB



BIG  
DATA

Preparation Program For  
AMCAT/e-Litmus/CoCubes/CRT

Campus Drive For Every  
Registered Students

# CRT



► Dedications Sessions On

- ✓ Logical Reasoning
- ✓ Verbal Ability
- ✓ Soft Skills
- ✓ Technical Skills

► Comprehensive

- ✓ Verbal & Soft Skills
- ✓ Sessions

► Assess Yourself

- ✓ First Time @ Gwalior
- ✓ Dedicated CRT Lab
- ✓ For Topic & Company Wise Assessments

► Authorised Placement Partner:



# Syllabus - Campus Recruitment Training (CRT)

## Description of Training on Quantitative Aptitude and Reasoning

- Number System
- Logarithms
- Average
- Problem on Ages
- Percentages
- Profit & Loss
- Ratio & Proportion
- Time & Work
- Pipes & Cisterns
- Time, Speed & Distance
- Allegations and Mixture
- Simple Interest & Compound Interest
- Permutation & Combination
- Probability
- Geometry

## LOGICAL REASONING

- Analogy /Series Completion
- Coding – Decoding
- Blood Relations
- Puzzle test
- Sitting Arrangement
- Direction Sense Test
- Logical Venn Diagrams
- Number, Ranking & Time Sequence Test
- Data Sufficiency
- Crypt arithmetic Questions

*10 Campus Drive For  
Registered Students*

## VERBAL ABILITY

- Reading Comprehension
- Jumbled Paragraph Questions
- Vocabulary Based Questions
- Fill in Blanks
- Miscellaneous Questions
- Error identification
- Error correction
- Antonyms
- Synonyms / Ordering sentences

**Course Fee: ₹ 9999/-**

### Special Features:-

1. Comprehensive Study Material/ Practice Sheets/ Test paper.
2. Online Mock test will be conduct after the completion of every topic.
3. 8 Diagnostic Career Test and 2 Pre-Assessment Test
4. Get directly interview calls from companies only after a good score in Pre-Assess.

## TECHNICAL INTERVIEW PREPARATION

### Topics to be covered in C

- Basic Concepts of C Language.
- Basic Programming Skills.
- Arrays, Pointers & Structures.

### Topics to be covered in C++:

- Introduction to C++
- Object-Oriented Programming Concepts
- The Basics of C++
- Pointers and Arrays
- Function and Operator Overloading
- Reusing classes
- Virtual functions and Polymorphism

### Topics to be covered in Datastructures:

- Introduction to datastructures.
- Stacks/Queue
- Linked list
- Tree
- Sorting Algorithms.

### Topics to be covered in Databases:

- Introduction to Database.
- Introduction to Normalization.
- Introduction to Data Definition Language/DML.
- Introduction to SQL/SQL Functions.
- Introduction to Set Operators, Groups, Reports.

## SOFT SKILLS

- Training need analysis (TNA)
- Behavioural training
- English language & Communication training
- Group Discussion
- Resume \ Curriculum vitae
- Interview Skills
- Email Etiquettes
- Business Etiquette and Customer Handling
- Etiquette of dressing
- Mock Group Discussions
- Mock Interviews
- Mock Recruitment Drive

PS-SOFTECH, Gwalior

# [Java-Part One]

[Java Basics, Class & Objects, Inheritance, Garbage Collection, Constructors, Exception Handling]

Compiled By Sandeep Sappal

# Index

<b>Introduction.....</b>	<b>3</b>
<b>Company C, C++ &amp; java.....</b>	<b>5</b>
<b>Java &amp; internet.....</b>	<b>5</b>
<b>Java feature.....</b>	<b>6</b>
<b>Java security.....</b>	<b>7</b>
<b>Data types.....</b>	<b>9</b>
<b>Structure of java program.....</b>	<b>10</b>
<b>Understanding java program.....</b>	<b>11</b>
<b>System.out.println().....</b>	<b>13</b>
<b>Selection statements (if statement).....</b>	<b>14</b>
<b>Loop (iterative statement).....</b>	<b>16</b>
<b>Switch &amp; case.....</b>	<b>19</b>
<b>Arrays.....</b>	<b>20</b>
<b>Jagged arrays.....</b>	<b>27</b>
<b>Strings.....</b>	<b>32</b>
<b>String methods.....</b>	<b>33</b>
<b>StringBuffer.....</b>	<b>47</b>
<b>Command line arguments.....</b>	<b>51</b>
<b>Introduction to java classes.....</b>	<b>52</b>
<b>Access modifier.....</b>	<b>52</b>
<b>Objects.....</b>	<b>54</b>
<b>Java methods.....</b>	<b>56</b>
<b>this object.....</b>	<b>57</b>
<b>Method overloading.....</b>	<b>61</b>
<b>Constructors.....</b>	<b>63</b>
<b>Parameterized constructors.....</b>	<b>64</b>
<b>Overloaded constructor.....</b>	<b>66</b>
<b>Destroying objects (Garbage collection).....</b>	<b>68</b>

Static member.....	70
Wrapper classes.....	73
Inheritance.....	76
Single inheritance.....	78
Multilevel inheritance.....	80
Hierarchical inheritance.....	82
Constructor in inheritance(Super Keyword).....	84
Method overriding (Super Keyword).....	85
Dynamic method dispatch (Run Time polymorphism).....	87
Abstract classes & method.....	89
Final keywords.....	94
Interfaces (Run Time Polymorphism).....	96
Difference between interfaces and classes.....	98
Packages.....	99
Creating packages.....	100
Exception handling.....	102
Java exception checked/unchecked Exception.....	104
Handling multiple catch block.....	105
Nested try block.....	107
Throw keyword.....	110
Throws keyword.....	111
User defined exception.....	113
Finally Block.....	115

## Introduction

Java is General purpose object oriented programming language developed by sun micro system of usa in 1991.

Java was designed for the development of software for consumer electronics devices like TV, Toaster, VCR, and such electronic machines.

This goal has strong impact on development team to make language more simple reliable and portable.

Member of sun Patrick Naughtan discovered that the existing language like C and C++ had limitation in terms of reliability and portability.

So they modeled their new language Java on the basis fo C and C++ but removed number of pitfalls that were considered as a source of problem and thus made java really simple reliable and portable.

### **Java Milestone :-**

Year	Development
1990	Sun Microsystems decided to develop special software that could manipulate consumer electronics devices (James Gosling)
1991	Embedded OOPS in Java and Named OAK.
1993	WWW Introduces Graphical environment, the green project team introduces applet programming for internet.
1994	The team developed a web browser called Hot java to Locate and run programs on Internet.
1995	OAK was Renamed java due to some legal snag, Many popular companies like Netscape and Microsoft support Java.
1996	Java established itself not only as a leader for internet programming but also for application programming.

*Note : Java is the first language that is not tied to any particular hardware or OS. Program develop in Java can be executed anywhere on any System.*

## Java and C :-

- Java doesn't Support

- ✓ goto, sizeof, typedef
- ✓ struct, union and enum
- ✓ Pointers
- ✓ Pre-processor
- ✓ Void arguments in functions.

**Java Supports :**

- ✓ binary, bitwise operator
- ✓ break, continue
- ✓ some loops for, while, do while
- ✓ switch and case, if statements

**Java & C++ :-**

**- Does not support**

- ✓ Operator over Loading
- ✓ Template class
- ✓ Multiple inheritance
- ✓ Global variable
- ✓ Header files

**- Java support**

- ✓ Class and objects
- ✓ Constructor, encapsulation, inheritance, polymorphism

**Table Comparing C, C++ and Java**

Feature	C	C++	Java
Paradigms	Procedural	Procedural, OOP, Generic Programming	OOP, Generic Programming (from Java 5)
Form of Compiled Source Code	Executable Native Code	Executable Native Code	Java byte code
Memory management	Manual	Manual	Managed, using a garbage collector
Pointers	Yes, very commonly used.	Yes, very commonly used, but some form of references available too.	No pointers; references are used instead.
Pre-processor	Yes	Yes	No
String Type	Character arrays	Character arrays, objects	Objects
Complex Data Types	Structures, unions	Structures, unions, classes	Classes
Inheritance	N/A	Multiple class inheritance	Single class inheritance, multiple interface implementation
Operator Overloading	N/A	Yes	No
Automatic coercions (forcefully)	Yes, with warnings if loss could occur	Yes, with warnings if loss could occur	Not at all if loss could occur; must cast explicitly
Variadic Parameters	Yes	Yes	No
Goto Statement	Yes	Yes	No

### Java and Internet

- Java strongly support internet because the first application program written in Java was HOT JAVA a web Browser to run applet on internet.

- It is useful for internet user to setup their web sites with applets which is distributed world wide on internet. Java also introduces swings a new technology which works better than applets.
- Introduces Servlet to communicate with server.
- The latest technology introduces EJB (Enterprise Java Beans) and J2EE (Java Enterprise Edition) which is used to create distributed components (E-commerce, E-business).
- To communicate with WWW Java code enabled with HTML tags thus user can add animation, graphics, games to make page more interactive and dynamic.

## **Java Feature**

### **Compiled and Interpreted :-**

Java has two stage system, First Java Compiler compiles source code into byte code. In second stage java interpreter generates machine code that can be directly executed by machine.

### **Platform Independent and Portable :-**

Java programs can be easily moved from one computer to another anywhere any time. Changes and upgrade in os, processor and system resource will not force any changes in Java programming. Java applets download from remote computer via internet and execute locally. A Java compiler generates byte code instruction that can be implemented on any Machine.

### **Object Oriented :-**

Java is true object oriented language almost everything in java is an object. All program code and data reside within the object and class.

### **Robust and Secure :-**

Java provides many safeguards to ensure reliable code.

It has good memory management system to allocate or dellocate location without using pointers. Uses Exception Handling tech. capture series of errors and eliminate any risk of crashing system.

Security becomes more important issue in java, because java programs run on internet, threat of viruses and abuse of system resources is everywhere, java systems verify all memory access but also ensure that no viruses communicate with an applet.

**Distributed :-**

Java Application can open and access remote objects on internet as easily as they can do in local system.

**Simple, small and Familiar :-**

Java is very simple and small, it's simple because no unstructured statement is allowed by java, even it doesn't uses pointer.

It is small because its program occupy less memory space.

**Multithreaded :-**

Multithreaded means handling multiple task simultaneously. Java Support multithreaded programs means we need not wait for application to finish one task before beginning another.

Ex : Listen to an Audio clip with Editing. Editing and printing, spell checking.

**Dynamic and extensible :-**

Dynamically linking new class libraries, methods and object.

Java programs support function written in other language such as C and C++, these functions are known as native method.

**Java Architecture and Security****JVM (Java Virtual Machine)**

The client application or OS must have java byte code interpreter to execute byte code instruction this interpreter is called JVM.

Java virtual machine interprets the byte code (compiled code) into native code in general java byte code files doesn't execute as fast as native code (C++ or C).

Every Machine or OS must have java interpreter to run java applets and application. In most of the cases the Java Interpreter is installed automatically when the end user installs a java compatibility web browser. Ex. IE 3.0, Netscape Navigator 2.0.

When browser invokes the JVM to run java applets the JVM does the following things.

- It validate requested byte code, verifying that they pass through various formatting and security checks, this security feature is known as byte code verifier.
- It allocates memory for incoming Java class files and guarantees that the security sandbox of JVM is not violated.
- It interpret byte code instructions found in the class files.
- It resolve any call of standard libraries called java core package.

## **JIT (Just In Time)**

Speed of execution can be issue for java larger program because java byte code are slower then native code execution. To overcome this problem java vendor include JIT compiler with their browsers or java compilers. This components plugs into java architecture by replacing or supplementing the java interpreter. As the name suggest instead of interpreting line by line, byte code instruction, JIT compiler convert entire java method into corresponding native code equivalents.

## **SECURITY MODEL OF JAVA**

### **Byte code verifier:-**

Before executing any class file the byte code verifier in the runtime system subjects them to a series of test.

- ✓ doesn't forge pointers( doesn't allow pointer Arithmetic)
- ✓ doesn't violate access restrictions
- ✓ call methods with appropriate arguments of the appropriate types.
- ✓ Avoid stack overflow
- ✓ Doesn't try to execute any insecure restriction.

### **Class Loader :-**

- ✓ The class loader allocate memory for each class.
- ✓ It also check the code written by user that it doesn't deceive or dislodge a built in class, means a programmer doesn't write her own version of class which already exist in java.

### **Sandbox model of Security :**

Sandbox determine the java applet can't do

- ✓ Delete file on local system.
- ✓ Read file from local system.
- ✓ Create new directories on the local system.
- ✓ Execute programs on the local system.

- ✓ Call DLL
- ✓ Create network connection to machine other then the server.
- ✓ Create object from the core package.

### **Types of Variable**

(Primitives data)

#### **INTEGERS TYPES**

Type	Size	Min. Value	Max. Value	Default Value
byte	1	-128	127	0
short	2	-32,768	32,767	0
int	4	-2,147,483,648	+2,147,483,647	0
long	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807	0l

#### **FLOATING TYPES**

Type	Size	Min. Value	Max. Value	Default Value
float	4 bytes	3.4e-038	3.4e+0.38	0.0f
double	8 bytes	1.7e-308	1.7e+308	0.0d

#### **CHARACTER TYPE**

Type	Size	Unicode	Default Value
char	2 bytes	unicode	null

#### **BOOLEAN TYPE**

Type	Size	Value	Default Value
boolean	1 bit	true, false	false

## Operators in Java

### Arithmetic

+, -, \*, / %

### Logical

&&, ||, !

### Relational

>, <, <=, >=, ==, !=

### Assignment

=, +=, /=, \*=, %=

### Unary

++, --

### Bitwise

&, |, ^, ~

## Back Slash Character

\b	backspace
\f	formfeed
\n	newline
\r	return key
\t	tab (space)
\'	print '

\"	print “
\\	print \

## Structure of Java Program

Document section	[optional]
Package statement	[optional]
Import statement	[optional]
Interface statement	[optional]
Class definition	[optional]

Main method class	[Mandatory]
{	
main method definition	
{       }	

### Syntax :-

```
class < class name >
{
    public static void main (string any [] )
    {
        Statement.....
    }
}
```

### About main :

**Public :** The keyword public is specifier which indicate that this function accessible to all other classes & user.

**Static :** Since main is static you don't need to create an instance (object) of the class in. order to call it. Main is called by Java

## Understanding Hello World Java Program

Before start hard programming in Java, its necessary to understand each and every part of the program. Lets understand the meaning of public, class, main, String[] args, System.out, and so on.

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

### **Class Declaration:**

Class is the building block in Java, each and every methods & variable exists within the class or object. (instance of program is called object ). The *public* word specifies the accessibility of the class. The visibility of the class or function can be public, private, etc. The following code declares a new class "HelloWorld" with the public accessibility:

```
public class HelloWorld {
```

### **The main Method:**

The main method is the entry point in the Java program and java program can't run without main method. JVM calls the main method of the class. This method is always first thing that is executed in a java program. Here is the main method:

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

{ and is used to start the beginning of main block and } ends the main block. Everything in the main block is executed by the JVM.

The code:

```
System.out.println("Hello, World");
```

prints the "Hello World" on the console. The above line calls the *println* method of *System.out* class.

**The keyword static:**

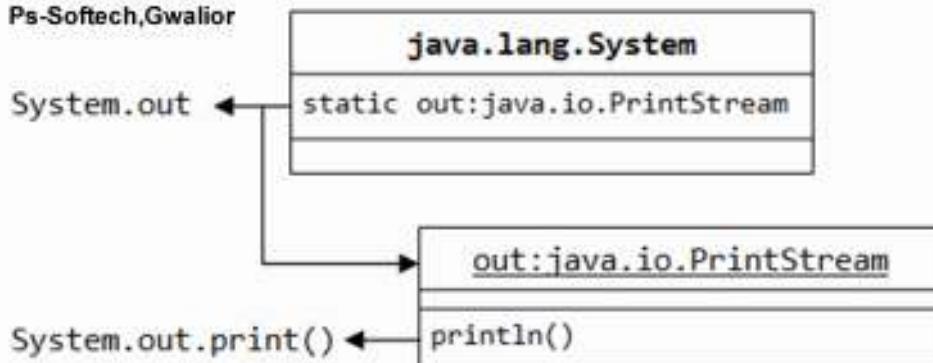
The keyword **static** indicates that the method is a *class* method, which can be called without the requirement to instantiate an object of the class. This is used by the Java interpreter to launch the program by invoking the **main** method of the class identified in the command to start the program.

**System.out.println()**

If you check the JDK API specification, you will find that:

- "System" is a class in the package `java.lang`.
- "out" is a static public variable of the class `java.lang.System`.
- "out" belongs a class called "`java.io.PrintStream`".
- The class `java.io.PrintStream` provides a public method called "`println()`".

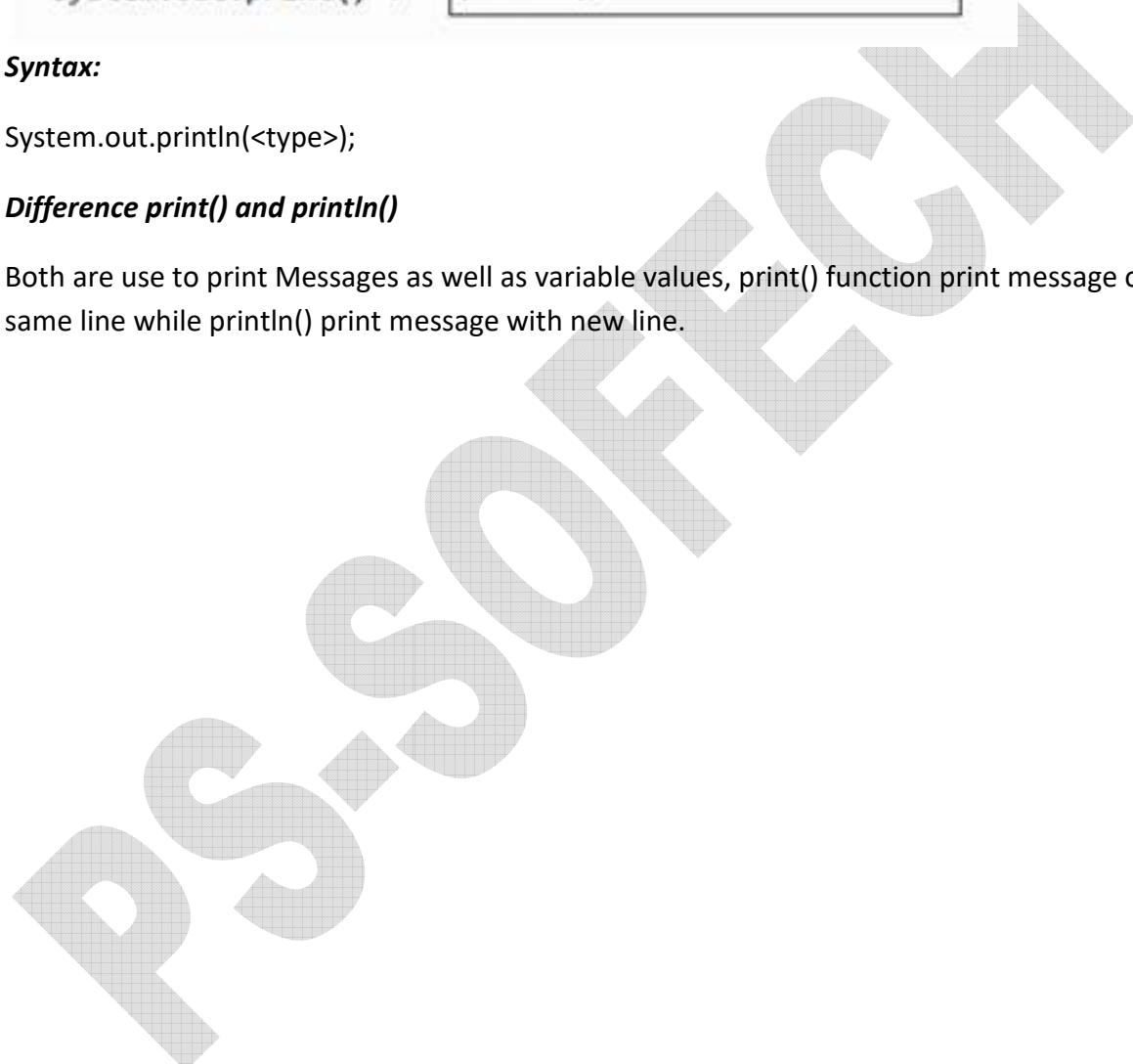
The following figure illustrate the classes involved in `System.out.println()`.

***Syntax:***

```
System.out.println(<type>);
```

***Difference print() and println()***

Both are used to print Messages as well as variable values, print() function prints message on same line while println() prints message with new line.

**Selection Statements**

*Conditional statements control the sequence of statement execution, depending on the value of an expression*

**Syntax**

```
if ( <boolean expression> ) {  
    <statement list>  
} else {  
    <statement list>  
}
```

The else clause is optional:

```
if ( <boolean expression> ) {  
    <statement list>  
}
```

### Semantics

- If the boolean expression is true, the first statement list is executed, and the second statement list (if any) is skipped.
- If the boolean expression is false, the first statement list is skipped, and the second statement list (if any) is executed.

### Nesting

If statements can be nested:

```
if ( <boolean expression> ) {  
    <statement list>  
} else if ( <boolean expression> ) {  
    <statement list>  
    ...  
} else if ( <boolean expression> ) {  
    <statement list>  
} else {  
    <statement list>  
}
```

### Example

```
*****  
* File: Conditional.java
```

- \* Purpose: Program to show conditional control with if statements
- \* Author: Sandeep Sappal

\*\*\*\*\*\*/

```
public class Conditional {
public static void main (String [] Argv) {
    int Value;

    Value = 0;
    if ((Value % 3) == 0) {
        System.out.println (Value + " is divisible by 3.");
    }
    if ((Value % 2) == 0) {
        System.out.println (Value + " is an even number.");
    } else {
        System.out.println (Value + " is an odd number.");
    }
    if (Value > 0) {
        System.out.println (Value + " is a positive number.");
    } else if (Value < 0) {
        System.out.println (Value + " is a negative number.");
    } else {
        System.out.println (Value + " is neither positive nor negative.");
    }
}
}
```

### **Output**

0 is divisible by 3.  
0 is an even number.

### **Iterative Statements**

## The while Statement

- Iterative statements repeated execution of list of statements, depending on the value of an expression
- The Java while statement repeats depending on the value of a boolean expression evaluated before each iteration of the loop.
- Syntax:  

```
while (<boolean expression>) {
    <statement list>
}
```
- The statement list (the body of the loop) may not be executed at all.
- As long as the boolean expression is true, the statement list is executed repeatedly.
- When the boolean expression is false, the statement list is skipped, and execution continues with the statement following the while statement.

## A Program using Iterative Statements

```
*****
* File: Iterative.java
* Purpose: Program to show iteration with a while loop
*****/
public class Iterative {
    public static void main (String [] Argv) {
        int Limit;
        int Counter;

        Limit = 4;
        Counter = 1;
        while (Counter <= Limit) {
            System.out.println ("Iteration " + Counter + " of " + Limit);
            Counter = Counter + 1;
        } }}

```

- Program output:

Iteration 1 of 4  
 Iteration 2 of 4  
 Iteration 3 of 4  
 Iteration 4 of 4

## The for Statement

- Iterative statements repeated execution of list of statements, depending on the value of an expression.
  - The Java for statement repeats depending on the value of a boolean expression evaluated before each iteration of the loop.
  - Syntax:
- ```
for (<expression 1>; <boolean expression>; <expression 2>) {
    <statement list>
}
```
- Expression 1 is evaluated before entering the loop.
  - As long as the boolean expression is true, the statement list is executed repeatedly.
  - Expression 2 is evaluated after the body of the loop.
  - When the boolean expression is false, the statement list is skipped, and execution continues with the statement following the for statement.
  - The statement list (the body of the loop) may not be executed at all.
  - The *for* statement is semantically equivalent to a *while* statement:

```
<expression 1>;
while (<boolean expression>) {
    <statement list>
    <expression 2>;
}
```

## A Program using Iterative Statements

```
*****
 * File: ForDemo.java
 * Purpose: Program to show iteration with a for loop
 ****/
public class ForDemo {
    public static void main (String [] Argv) {
        int Limit;
        int Counter;
        Limit = 4;
        for (Counter = 1; Counter <= Limit; Counter = Counter + 1) {
            System.out.println ("Iteration " + Counter + " of " + Limit);
        }
    }
}
```

Program output:

```
Iteration 1 of 4
Iteration 2 of 4
Iteration 3 of 4
Iteration 4 of 4
```

## The do Statement

- Iterative statements repeated execution of list of statements, depending on the value of an expression
- The Java do ... while statement repeats depending on the value of a boolean expression evaluated after each iteration of the loop.
- Syntax:  

```
do {
    <statement list>
} while ( <boolean expression> );
```
- The *while* clause is followed by a semicolon (;).
- The statement list (the body of the loop) is executed at least once.
- As long as the boolean expression is true, the statement list is executed repeatedly.
- When the boolean expression is false, execution continues with the statement following the *do* statement.

## A Program using Iterative Statements

```
*****
* File: DoWhileDemo.java
* Purpose: Program to show iteration with a do ... while loop
*****/
public class DoWhileDemo {
    public static void main (String [] Argv) {
        int Limit;
        int Counter;

        Limit = 4;
        Counter = 1;
        do {
            System.out.println ("Iteration " + Counter + " of " + Limit);
            Counter = Counter + 1;
        } while (Counter <= Limit);
    }
}
```

Program output:

```
Iteration 1 of 4
Iteration 2 of 4
Iteration 3 of 4
Iteration 4 of 4
```

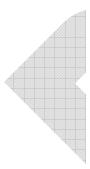
## The switch Statement

- Conditional statements control the sequence of statement execution, depending on the value of an expression
- The Java *switch* statement is controlled by an integer or char expression.
- Syntax:  

```
switch ( <integer expression> ) {
    case <value>:
        <statement list>
        break;
    case <value>:
        <statement list>
        break;
    ...
    default:
        <statement list>
        break;
}
```
- Control is transferred to the *case* according to the value of the control expression.
- The default clause is executed if the value of the control expression does not match any of the case values.

## A Program Using a Switch Statement

```
*****
File: Switcher.java
 * Purpose: Program to show conditional control with switch statements
*****
public class Switcher {
    public static void main (String [] Argv) {
        int Value;
        Value = 479;
        switch (Value % 4) {
            case 0:
                System.out.println (Value + " is divisible by 4.");
                break;
            case 2:
                System.out.println (Value + " is divisible by 2.");
                break;
            default:
                System.out.println (Value + " is an odd number.");
                break; } }}
```



Program output:  
479 is an odd number.

## Arrays

**Array:** Array is the most important thing in any programming language. By definition, array is the static memory allocation. It allocates the memory for the same data type in sequence. It contains multiple values of same types. It also stores the values in memory at the fixed size. Multiple types of arrays are used in any programming language such as: one - dimensional, two - dimensional or can say multi - dimensional.

### Array declarations

- arrays are Java objects
- they allocate their memory at run time using new operator
- they are reference variable and holds base address of memory location
- all Java arrays are technically one-dimensional. Two-dimensional arrays are arrays of arrays.
- array variable declarations must indicate a dimension by using []
- attempting to access an array with an index less than 0 or greater than the length of the array causes an **ArrayIndexOutOfBoundsException** to be thrown at runtime
- since arrays are Objects they can be initialized using the **new** operator
- when created, arrays are automatically initialized with the default value of their type
- **Length** is a JRE Constant which returns size of an array

### Java Array Default Values

After you instantiate an array, default values are automatically assigned to it in the following manner.

- byte – default value is zero
- short – default value is zero
- int – default value is zero
- long – default value is zero, 0L.
- float – default value is zero, 0.0f.
- double – default value is zero, 0.0d.
- char – default value is null, '\u0000'.
- boolean – default value is false.
- reference types – default value is null.

***Using and array in your program is a 3 step process -***

- 1) Declaring your Array
- 2) Constructing your Array
- 3) Initializing your Array

**Syntax for Declaring Array Variables is**

```
<elementType>[] <arrayName>;
```

or

```
<elementType> <arrayName>[];
```

Example:

```
int X[]; // Defines that X is an ARRAY variable of which will store integer values
```

```
int []X;
```

**Constructing an Array**

```
<arrayName> = new <elementType>[<noOfElements>];
```

Example:

```
X = new int[10]; // Defines that X will store 10 integer values
```

**Declaration and Construction combined**

```
int X[] = new int[10];
```

**Initializing an Array**

```
X[0]=1; // Assigns an integer value 1 to the first element 0 of the array
```

```
X[1]=2; // Assigns an integer value 2 to the second element 1 of the array
```

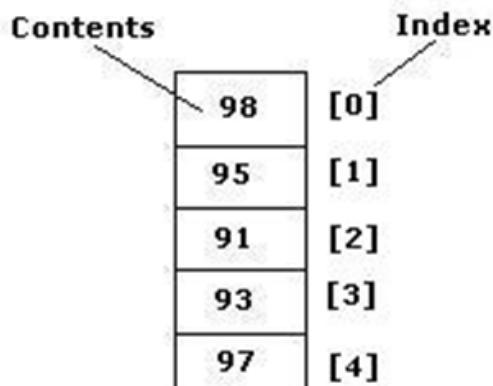
### Declaring and Initializing an Array

```
<elementType>[] <arrayName> = {<arrayInitializerCode>};
```

Example:

```
int X[] = {98,95,91,93, 97}; // Initializes an integer array of length 5 where the first element is 0,1 , second element is 2 and so on.
```

This is the logical view of Above Example



Logical View of an Array

PS-Softech,Gwalior

But when arrays allocate their memory locations using new operator or while at the time of initialization it will physically stored like this as in following diagram.

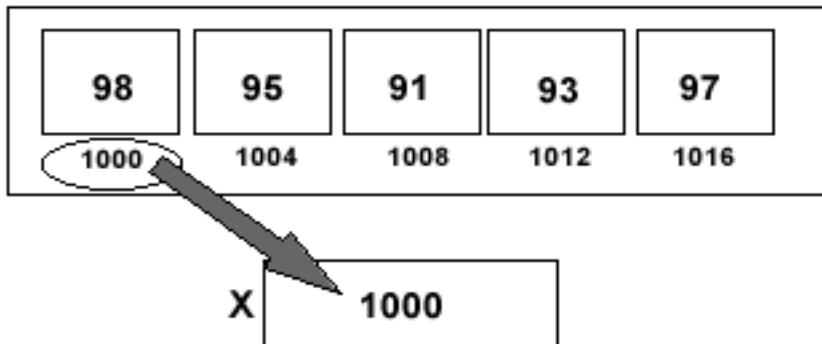
**Note: array object always holds reference(base address) of memory location.**

Here in diagram X is a reference variable holds base address of array heap i.e. 1000, now how to simplify X[1] using reference variable.

- X holds address 1000
- X+1 means proceed to next address i.e 1004
- [] Symbol represent Value at address 1004 i.e.95

## Physical Location of an Array

Array Heap

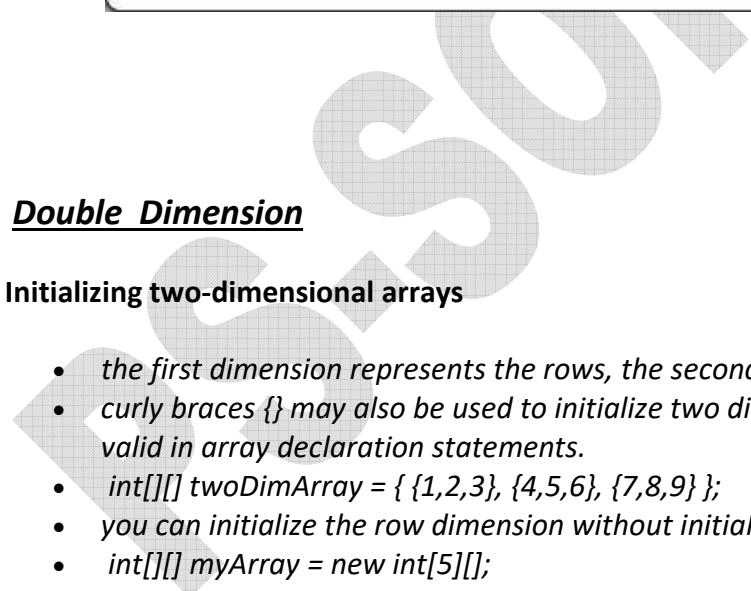


**X is a Reference Object Which Hold Base Address of Array Heap**

Ps-Softech Gwalior

```
*****
* File: exsing.java
* Purpose: Program to copy value of array X into Y
*****/
import java.util.*;
class exsing
{
    public static void main(String arg[])
    { Scanner KB=new Scanner(System.in);
        System.out.print("Enter No of Integers for Array X:");
        int n=KB.nextInt();
        int x[]=new int[n];
        for(int i=0;i<x.length;i++)
        {System.out.print("Enter ["+i+"]:");
        x[i]=KB.nextInt();
        }
        int y[]=new int[x.length];
        System.out.println("Values Copied in Y array are:");
        for(int i=0;i<x.length;i++)
        { y[i]=x[(x.length-1)-i];
        System.out.println(y[i]);
        }
    }
}
```

```
}
```



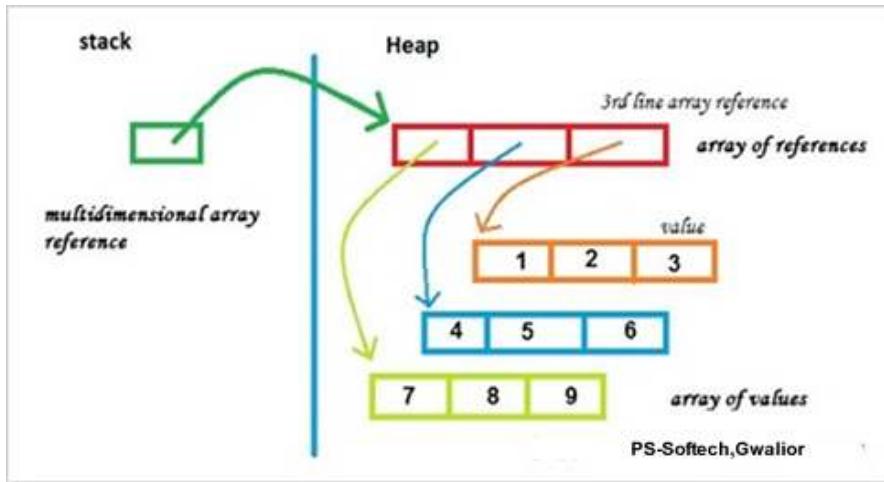
```
C:\Windows\system32\cmd.exe
Enter No of Integers for Array X:4
Enter [0]:23
Enter [1]:45
Enter [2]:67
Enter [3]:7
Values Copied in Y array are:
7
67
45
23
Press any key to continue . . .
```

## Double Dimension

### Initializing two-dimensional arrays

- *the first dimension represents the rows, the second dimension, the columns*
- *curly braces {} may also be used to initialize two dimensional arrays. Again they are only valid in array declaration statements.*
- *int[][] twoDimArray = { {1,2,3}, {4,5,6}, {7,8,9} };*
- *you can initialize the row dimension without initializing the columns but not vice versa*
- *int[][] myArray = new int[5][];*
  
- *int[][] myArray = new int[][5]; // illegal*
- *the length of the columns can vary*

## Array of Array



Now lets Take an Example of array Having 3 Rows and 4 Columns

Ex int [][]A=new int[3][4];

```
*****
* File: exdd.java
* Purpose: Program to input matrix & print it
*****
```

```
import java.util.*;
class exdd
{ public static void main(String arg[])
{ Scanner KB=new Scanner(System.in);
System.out.print("Enter Rows:");
int r=KB.nextInt();
System.out.print("Enter Cols:");
int c=KB.nextInt();
int A[][]=new int[r][c];
for(int i=0;i<A.length;i++)
{for(int j=0;j<A[i].length;j++)
{System.out.print("A["+i+"]["+j+"]:");
A[i][j]=KB.nextInt();}}
```

}}

```

for(int i=0;i<A.length;i++)
{ for(int j=0;j<A[i].length;j++)
{ System.out.print(A[i][j]+" ");
}
System.out.println(); } }

```

```

C:\Windows\system32\cmd.exe
Enter Rows :3
Enter Cols :4
A[0][0]:1
A[0][1]:0
A[0][2]:12
A[0][3]:-1
A[1][0]:7
A[1][1]:-3
A[1][2]:2
A[1][3]:5
A[2][0]:-5
A[2][1]:-2
A[2][2]:2
A[2][3]:9
1 0 12 -1
7 -3 2 5
-5 -2 2 9
Press any key to continue . . .

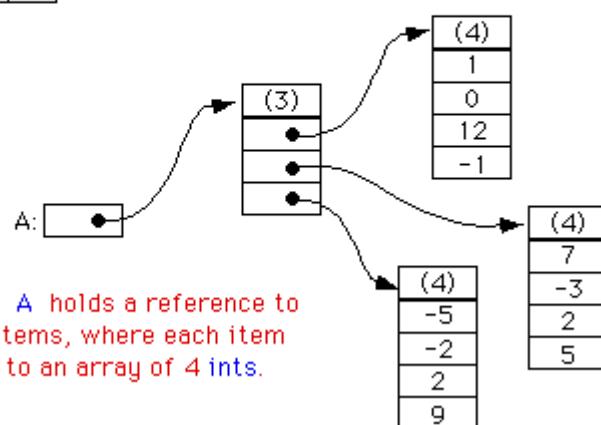
```

*A is a reference variable holds base address of memory block as in following diagram*

|    |    |    |    |    |
|----|----|----|----|----|
| A: | 1  | 0  | 12 | -1 |
|    | 7  | -3 | 2  | 5  |
|    | -5 | -2 | 2  | 9  |

If you create an array `A = new int[3][4]`,  
you should think of it as a "matrix" with  
3 rows and 4 columns.

But in reality, `A` holds a reference to  
an array of 3 items, where each item  
is a reference to an array of 4 ints.



1. **A.length** gives the number of rows present in the two-dimensional array. i.e. 3
2. **A[0].length** gives the number of elements present in the first row. i.e. 4
3. There is no predefined way to find the total number of elements present in a two-dimensional array.
4. **A[0][0]** returns the element existing in the first row and first column.i.e.1

## Jagged Array-Vary Column Size Array

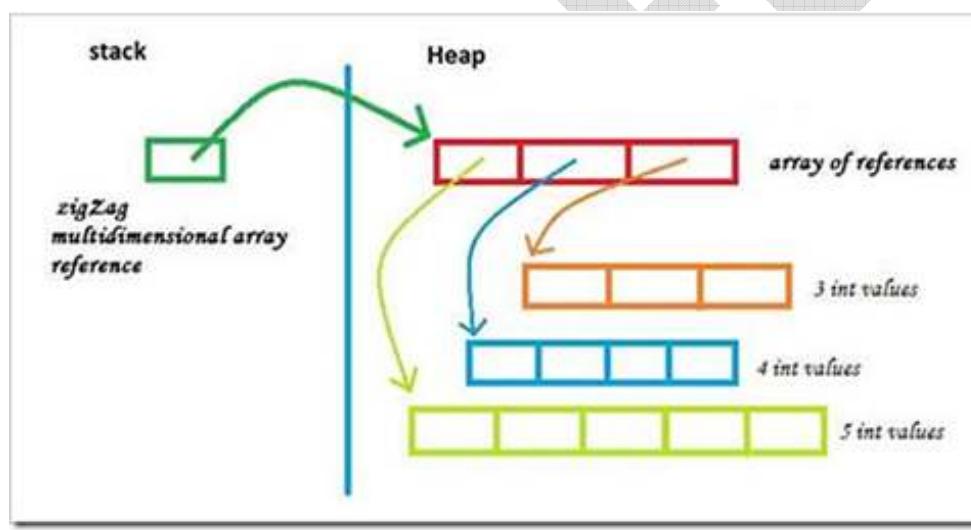
It is a new feature supported by Java. In **Jagged arrays**, each row, in a two-dimensional array, may contain different lengths. Let us design a two-dimensional array with 4 rows where the first row contains 4 elements, the second row with 1 element, the third row with 2 elements and the fourth row with 3 elements.

### Initialization

```
int A[][]={{1,2,3},{1,2},{1,2,3,4}};
```

### Declaration

```
int A[][]=new int[4][];
```



PS-Softech,Gwalior

Lets take an Example of Students Queue Having 4 Rows and each row Contains n Students

| Columns | 0  | 1  | 2  | 3  |
|---------|----|----|----|----|
| 0       | 44 | 55 | 66 | 77 |
| 1       | 36 |    |    |    |
| 2       | 87 | 97 |    |    |
| 3       | 68 | 78 | 88 |    |

Figure : Varying columns 2D array - matrix form

```
*****
* File: JaggedArray.java
* Purpose: Program print Student Queue with n cols
*****/
public class JaggedArrays
{
    public static void main(String args[])
    {
        int student[][] = new int[4][];
        student[0] = new int[4];
        student[1] = new int[1];
        student[2] = new int[2];
        student[3] = new int[3];
        System.out.println("Row count: " + student.length);
        System.out.println("Third row size: : " + student[3].length);
        // 1st row
        student[0][0] = 44;
        student[0][1] = 55;
        student[0][2] = 66;
        student[0][3] = 77;
        // 2nd row
        student[1][0] = 36;
        // 3rd row
```

```

student[2][0] = 87;
student[2][1] = 97;
        // 4th row
student[3][0] = 68;
student[3][1] = 78;
student[3][2] = 88;

System.out.println("student[3][1] marks: " + student[3][1]);

System.out.println("\nMatrix Form");

for(int i = 0; i < student.length; i++)
{
    for(int j = 0; j < student[i].length; j++)
    {
        System.out.print(student[i][j] + "\t");
    }
    System.out.println();
} }}

```

A screenshot of a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The command 'java JaggedArrays' is run, followed by output showing the row count (4), third row size (3), and the value at student[3][1] (78). Below this, the matrix form of the array is printed:

```

C:\snr\arrays>java JaggedArrays
Row count: 4
Third row size: : 3
student[3][1] marks: 78

Matrix Form
44      55      66      77
36
87      97
68      78      88

```

In the **student** array, rows are 4 and the columns are missing; it is done knowingly as each row should contain different lengths. Java allows to assign the row length for each individually as follows.

```

student[0] = new int[4];
student[1] = new int[1];
student[2] = new int[2];
student[3] = new int[3];

```

In the above four statements, sizes are given differently. The lengths of 4 rows are 4, 1, 2 and 3. This assignment for each row is not possible in C/C++. Values are assigned separately for each row elements.

## Tripple Dimension

### Java 3D Array

In Java, creating a 3-dimensional array is as simple as saying:

```
int[][][]a = new int[3][4][5];
```

**PS-Softech Gwalior**

```
int a[][][] = new int[3][4][3];
    ↑
    no of matrix is 3
    ↓
    4 rows in each matrix
    ↓
    3 Cols in each matrix
```

```
*****
* File: td.java
* Purpose: Program print input n matrix with n row and n cols and print it
*****  

import java.util.*;  

class td  

{ public static void main(String arg[])  

{ Scanner X=new Scanner(System.in);  

System.out.print("Enter No. of Matrix:");  

int n=X.nextInt();  

System.out.print("Enter No. of Rows:");  

int r=X.nextInt();  

System.out.print("Enter No. of Cols:");  

int c=X.nextInt();  

int a[][][] =new int[n][r][c];  

//read value thru keyboard  

for(int k=0;k<a.length;k++)  

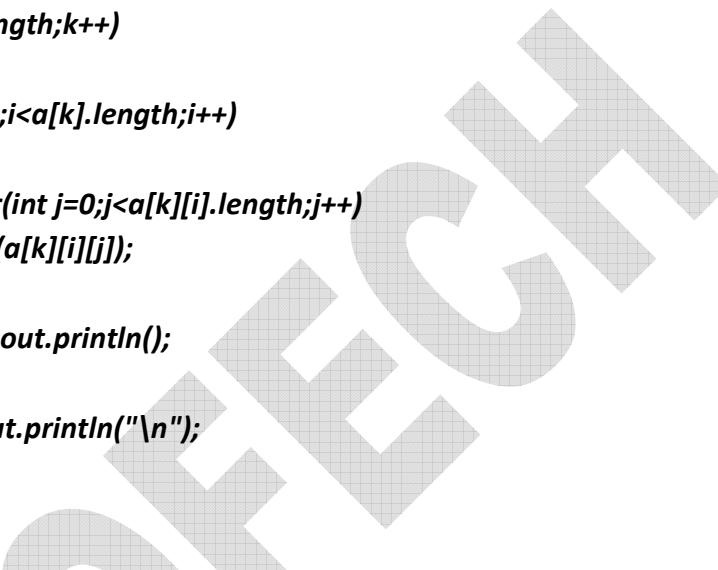
{ for(int i=0;i<a[k].length;i++)  

{ for(int j=0;j<a[k][i].length;j++)  

{System.out.print("Enter a["+k+"]["+i+"]["+j+"]:");
```

```
a[k][i][j]=X.nextInt();
    }
    System.out.println();
}
System.out.println("\n");

////display matrix
for(int k=0;k<a.length;k++)
{
    for(int i=0;i<a[k].length;i++)
    {
        for(int j=0;j<a[k][i].length;j++)
        {System.out.print(a[k][i][j]);
        }
        System.out.println();
    }
    System.out.println("\n");
} }}
```



```
C:\Windows\system32\cmd.exe
Enter No. of Matrix:2
Enter No. of Rows:2
Enter No. of Cols:2
Enter a[0][0][0]:1
Enter a[0][0][1]:1

Enter a[0][1][0]:2
Enter a[0][1][1]:2

Enter a[1][0][0]:3
Enter a[1][0][1]:3

Enter a[1][1][0]:4
Enter a[1][1][1]:4

11
22

33
44
```

## Strings

### **1. Introduction**

In Java strings are objects designed to represent a sequence of characters. Because character strings are commonly used in programs, Java supports the ability to declare String constants and perform concatenation of Strings directly without requiring access to methods of the String class. This additional support provided for Java Strings allows programmers to use Strings in a similar manner as other common programming languages.  
A Java String is read-only and once created the contents cannot be modified.

### **2. Declaring and Allocating Strings**

The String class has six different constructors. Only the most common constructors will be presented.

The simplest method to create a String object is to enclose the string literal in quotes and assign the value to a String object. Creation of a String through assignment does not require the use of the new operator, for example

```
String str = "abc";  
String language = "Java";
```

**Constructors:**

Alternatively, String objects can be created through constructors. The following constructors are supported:

**public String()**

Constructs a new String with the value "" containing no characters. The value is not null.

**public String( String value )**

Constructs a new String that contains the same sequence of characters as the specified String argument.

**public String( char[] value )**

Constructs a new String containing the same sequence of characters contained in the character array argument.

**public String(char[] value, int offset, int count)**

Allocates a new String that contains characters from a subarray of the character array argument.

**public String(byte[] bytes)**

Constructs a new String by decoding the specified array of bytes

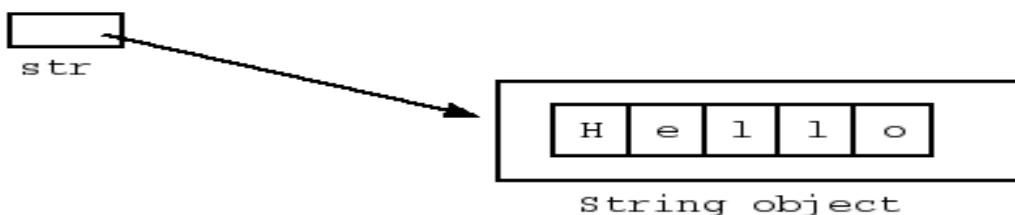
**public String(byte[] bytes, int position, int length)**

Constructs a new String by decoding the bytes, this method read length of bytes from position

**Example**

```
String s1 = new String();
String s2 = new String( "abc" ); // "abc" is first converted to a String and then
// passed as an String argument
String s3 = new String( s2 );
char data[] = { 'a', 'b', 'c' };
String s4 = new String( data );
```

```
String str = new String("Hello");
```



**String Methods:**

Here is the list methods supported by String class:

| <b>SN</b> | <b>Methods with Description</b>                                                                                                                               |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1         | <code>char charAt(int index)</code><br>Returns the character at the specified index.                                                                          |
| 2         | <code>int compareTo(Object o)</code><br>Compares this String to another Object.                                                                               |
| 3         | <code>int compareTo(String anotherString)</code><br>Compares two strings lexicographically.                                                                   |
| 4         | <code>int compareIgnoreCase(String str)</code><br>Compares two strings lexicographically, ignoring case differences.                                          |
| 5         | <code>String concat(String str)</code><br>Concatenates the specified string to the end of this string.                                                        |
| 6         | <code>static String copyValueOf(char[] data)</code><br>Returns a String that represents the character sequence in the array specified.                        |
| 7         | <code>static String copyValueOf(char[] data, int offset, int count)</code><br>Returns a String that represents the character sequence in the array specified. |
| 8         | <code>boolean endsWith(String suffix)</code><br>Tests if this string ends with the specified suffix.                                                          |
| 9         | <code>boolean equals(Object anObject)</code><br>Compares this string to the specified object.                                                                 |
| 10        | <code>boolean equalsIgnoreCase(String anotherString)</code><br>Compares this String to another String, ignoring case considerations.                          |
| 11        | <code>byte getBytes()</code><br>Encodes this String into a sequence of bytes using the platform's default charset, storing                                    |

|    |                                                                                                                                                                                                  |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | the result into a new byte array.                                                                                                                                                                |
| 12 | <u>int indexOf(int ch)</u><br>Returns the index within this string of the first occurrence of the specified character.                                                                           |
| 13 | <u>int indexOf(String str)</u><br>Returns the index within this string of the first occurrence of the specified substring.                                                                       |
| 14 | <u>int indexOf(String str, int fromIndex)</u><br>Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.                       |
| 15 | <u>int lastIndexOf(int ch)</u><br>Returns the index within this string of the last occurrence of the specified character.                                                                        |
| 16 | <u>int lastIndexOf(int ch, int fromIndex)</u><br>Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.     |
| 17 | <u>int lastIndexOf(String str)</u><br>Returns the index within this string of the rightmost occurrence of the specified substring.                                                               |
| 18 | <u>int lastIndexOf(String str, int fromIndex)</u><br>Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. |
| 19 | <u>int length()</u><br>Returns the length of this string.                                                                                                                                        |
| 20 | <u>String replace(char oldChar, char newChar)</u><br>Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.                                       |
| 21 | <u>String[] split(String regex)</u><br>Splits this string around matches of the given regular expression.                                                                                        |

|    |                                                                                                                                              |
|----|----------------------------------------------------------------------------------------------------------------------------------------------|
| 22 | <u>boolean startsWith(String prefix)</u><br>Tests if this string starts with the specified prefix.                                           |
| 23 | <u>String substring(int beginIndex)</u><br>Returns a new string that is a substring of this string.                                          |
| 24 | <u>String substring(int beginIndex, int endIndex)</u><br>Returns a new string that is a substring of this string.                            |
| 25 | <u>char[] toCharArray()</u><br>Converts this string to a new character array.                                                                |
| 26 | <u>String toLowerCase()</u><br>Converts all of the characters in this String to lower case using the rules of the default locale.            |
| 27 | <u>String toLowerCase(Locale locale)</u><br>Converts all of the characters in this String to lower case using the rules of the given Locale. |
| 28 | <u>String toString()</u><br>This object (which is already a string!) is itself returned.                                                     |
| 29 | <u>String toUpperCase()</u><br>Converts all of the characters in this String to upper case using the rules of the default locale.            |
| 30 | <u>String trim()</u><br>Returns a copy of the string, with leading and trailing whitespace omitted.                                          |

✓ **charAt ()**

**Description:**

This method returns the character located at the String's specified index. The string indexes start from zero.

**Syntax:**

Here is the syntax of this method:

```
public char charAt(int index)
```

**Parameters:**

Here is the detail of parameters:

- **index** : Index of the character to be returned.

**Return Value :**

- Returns a char at the specified index.

**Example:**

```
public class Test{
    public static void main(String args[]){
        String s = "Strings are immutable";
        char result = s.charAt(8);
        System.out.println(result);  }}
```

This produces following result:

A

✓ *int compareTo(Object o)*

**Description:**

There are two variant of this method. First method compares this String to another Object and second method compares two strings lexicographically.

**Syntax:**

Here is the syntax of this method:

```
int compareTo(Object o)
```

or

```
int compareTo(String anotherString)
```

#### Parameters:

Here is the detail of parameters:

- **Object o** : the Object to be compared.
- **String anotherString** : the String to be compared.

#### Return Value :

- The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

#### Example:

```
public class Test{  
    public static void main(String args[]){  
        String str1 = "Strings are immutable";  
        String str2 = "Strings are immutable";  
        String str3 = "Integers are not immutable";  
  
        int result = str1.compareTo( str2 );  
        System.out.println(result);  
  
        result = str2.compareTo( str3 );  
        System.out.println(result);  
  
        result = str3.compareTo( str1 );  
        System.out.println(result);  }}
```

This produces following result:

|     |
|-----|
| 0   |
| 10  |
| -10 |

✓ **public String concat(String s)**

#### Description:

This method appends one String to the end of another. The method returns a String with the value of the String passed in to the method appended to the end of the String used to invoke this method.

#### Syntax:

Here is the syntax of this method:

```
public String concat(String s)
```

#### Parameters:

Here is the detail of parameters:

- **String str** : the String that is concatenated to the end of this String.

#### Return Value :

- A string that represents the concatenation of this object's characters followed by the string argument's characters.

#### Example:

```
public class Test{  
    public static void main(String args[]){  
        String s = "Strings are immutable";  
        s = s.concat(" all the time");  
        System.out.println(s); }}
```

This produces following result:

```
Strings are immutable all the time
```

✓ **public boolean equals(Object anObject)**

**Description:**

This method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

**Syntax:**

Here is the syntax of this method:

```
public boolean equals(Object anObject)
```

**Parameters:**

Here is the detail of parameters:

- **anObject** : the object to compare this String against.

**Return Value :**

- true if the String are equal; false otherwise.

**Example:**

```
public class Test{  
    public static void main(String args[]){  
        String Str1 = new String("This is really not immutable!!");  
        String Str2 = Str1;  
        String Str3 = new String("This is really not immutable!!");  
        boolean retVal;  
  
        retVal = Str1.equals( Str2 );  
        System.out.println("Returned Value " + retVal );  
  
        retVal = Str1.equals( Str3 );  
        System.out.println("Returned Value " + retVal );  
    }  
}
```

This produces following result:

```
Returned Value true  
Returned Value true
```

✓ ***indexOf()***

**Description:**

This method has following different variants:

- **public int indexOf(int ch):** Returns the index within this string of the first occurrence of the specified character or -1 if the character does not occur.
- **public int indexOf(int ch, int fromIndex):** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index or -1 if the character does not occur.
- **int indexOf(String str):** Returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.
- **int indexOf(String str, int fromIndex):** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. If it does not occur, -1 is returned.

**Syntax:**

Here is the syntax of this method:

```
public int indexOf(int ch )  
or  
public int indexOf(int ch, int fromIndex)  
or  
int indexOf(String str)  
or  
int indexOf(String str, int fromIndex)
```

**Parameters:**

Here is the detail of parameters:

- **ch:** a character.
- **fromIndex:** the index to start the search from.
- **str :** A string.

**Return Value :**

- See the description.

**Example:**

```
import java.io.*;
public class Test{
    public static void main(String args[]){
        String Str = new String("Welcome to Tutorialspoint");
        String SubStr1 = new String("Tutorials" );
        String SubStr2 = new String("Sutorials" );

        System.out.print("Found Index :");
        System.out.println(Str.indexOf( 'o' ));
        System.out.print("Found Index :");
        System.out.println(Str.indexOf( 'o', 5 ));
        System.out.print("Found Index :");
        System.out.println( Str.indexOf( SubStr1 ) );
        System.out.print("Found Index :");
        System.out.println( Str.indexOf( SubStr1, 15 ) );
        System.out.print("Found Index :");
        System.out.println( Str.indexOf( SubStr2 ) );
    }
}
```

This produces following result:

|                 |
|-----------------|
| Found Index :4  |
| Found Index :9  |
| Found Index :11 |
| Found Index :-1 |
| Found Index :-1 |

**public String[] split(String regex)**

#### Description:

This method has two variants and splits this string around matches of the given regular expression.

#### Syntax:

Here is the syntax of this method:

```
public String[] split(String regex, int limit)
or
public String[] split(String regex)
```

#### Parameters:

Here is the detail of parameters:

- **regex** : the delimiting regular expression.
- **limit** : the result threshold which means how many strings to be returned.

#### Return Value :

- It returns the array of strings computed by splitting this string around matches of the given regular expression.

#### Example:

```
import java.io.*;
public class Test{
    public static void main(String args[]){
        String Str = new String("Welcome-to-Tutorialspoint");

        System.out.println("Return Value :");
        for (String retval: Str.split("-", 2)){
            System.out.println(retval);
        }
        System.out.println("");
        System.out.println("Return Value :");
        for (String retval: Str.split("-", 3)){

```

```
    System.out.println(retval);
}
System.out.println("");
System.out.println("Return Value : ");
for (String retval: Str.split("-", 0)){
    System.out.println(retval);
}
System.out.println("");
System.out.println("Return Value : ");
for (String retval: Str.split("-")){
    System.out.println(retval);
}
}
```

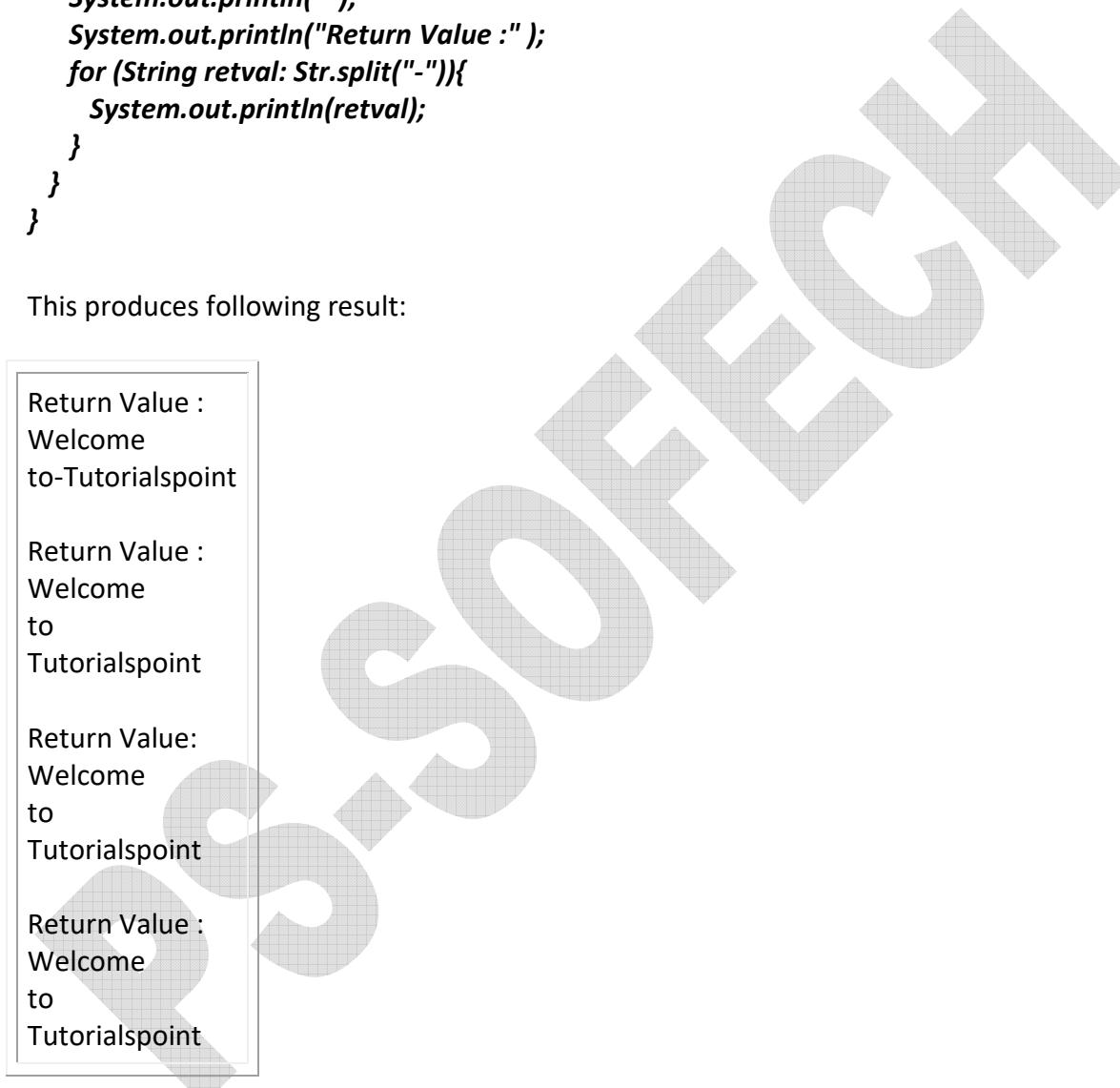
This produces following result:

```
Return Value :
Welcome
to-Tutorialspoint

Return Value :
Welcome
to
Tutorialspoint

Return Value:
Welcome
to
Tutorialspoint

Return Value :
Welcome
to
Tutorialspoint
```

A large, semi-transparent watermark reading "Tutorialspoint" diagonally across the page.

✓ ***public String substring(int beginIndex)***

#### Description:

This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or upto endIndex - 1 if second argument is given.

#### Syntax:

Here is the syntax of this method:

```
public String substring(int beginIndex)  
or  
public String substring(int beginIndex, int endIndex)
```

#### Parameters:

Here is the detail of parameters:

- **beginIndex** : the begin index, inclusive.
- **endIndex** : the end index, exclusive.

#### Return Value :

- The specified substring.

#### Example:

```
import java.io.*;  
public class Test{  
    public static void main(String args[]){  
        String Str = new String("Welcome to Tutorialspoint");  
  
        System.out.print("Return Value :" );  
        System.out.println(Str.substring(10));  
  
        System.out.print("Return Value :" );
```

```

        System.out.println(Str.substring(10, 15));
    }
}

```

This produces following result:

|                               |
|-------------------------------|
| Return Value : Tutorialspoint |
| Return Value : Tuto           |

#### ✓ ***public boolean startsWith(String prefix)***

#### **Description:**

This method has two variants and tests if a string starts with the specified prefix beginning a specified index or by default at the beginning.

#### **Syntax:**

Here is the syntax of this method:

|                                                       |
|-------------------------------------------------------|
| public boolean startsWith(String prefix, int toffset) |
|-------------------------------------------------------|

or

|                                          |
|------------------------------------------|
| public boolean startsWith(String prefix) |
|------------------------------------------|

#### **Parameters:**

Here is the detail of parameters:

- **prefix** : the prefix to be matched.
- **toffset** : where to begin looking in the string.

#### **Return Value :**

- It returns true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise.

#### **Example:**

```
import java.io.*;
```

```
public class Test{  
    public static void main(String args[]){  
        String Str = new String("Welcome to Tutorialspoint");  
  
        System.out.print("Return Value :");  
        System.out.println(Str.startsWith("Welcome"));  
  
        System.out.print("Return Value :");  
        System.out.println(Str.startsWith("Tutorials"));  
  
        System.out.print("Return Value :");  
        System.out.println(Str.startsWith("Tutorials", 11));  
    }  
}
```

This produces following result:

```
Return Value :true  
Return Value :false  
Return Value :true
```

## **StringBuffer**

The **StringBuffer** class is used to represent characters that can be modified. This is simply used for concatenation or manipulation of the strings.

**StringBuffer** is mainly used for the dynamic string concatenation which enhances the performance. A string buffer implements a mutable sequence of characters. A string buffer is like a **String**, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls. There are some functions used in the given example. All the functions have been explained below with example :

### **Creation of StringBuffers**

#### **StringBuffer Constructors**

- ✓ **StringBuffer(String): Construct String With Capacity of 16**
- ✓ **StringBuffer(int Capacity) : With capacity 100**
- ✓ **StringBuffer(): Default Capacity 16**

```
public class StringBufferDemo {

    public static void main(String[] args) {
        //      Examples of Creation of Strings
        StringBuffer strBuf1 = new StringBuffer("Bob");
        StringBuffer strBuf2 = new StringBuffer(100); //With capacity 100
        StringBuffer strBuf3 = new StringBuffer(); //Default Capacity 16
        System.out.println("strBuf1 : " + strBuf1);
        System.out.println("strBuf2 capacity : " + strBuf2.capacity());
        System.out.println("strBuf3 capacity : " + strBuf3.capacity());
    }
}
```

## Output

```
strBuf1 : Bob  
strBuf2 capacity : 100  
strBuf3 capacity : 16
```

## StringBuffer Functions

The following program explains the usage of some of the basic StringBuffer methods like

### 1. **capacity()**

Returns the current capacity of the String buffer.

### 2. **length()**

Returns the length (character count) of this string buffer.

### 3. **charAt(int index)**

The specified character of the sequence currently represented by the string buffer, as indicated by the index argument, is returned.

### 4. **setCharAt(int index, char ch)**

The character at the specified index of this string buffer is set to ch

### 5. **toString()**

Converts to a string representing the data in this string buffer

### 6. **insert(int offset, char c)**

Inserts the string representation of the char argument into this string buffer.

Note that the StringBuffer class has got many overloaded 'insert' methods which can be used based on the application need.

### 7. **delete(int start, int end)**

Removes the characters in a substring of this StringBuffer

### 8. **replace(int start, int end, String str)**

Replaces the characters in a substring of this StringBuffer with characters in the specified String.

### 9. **reverse()**

The character sequence contained in this string buffer is replaced by the reverse of the sequence.

### 10. **append(String str)**

Appends the string to this string buffer.

Note that the StringBuffer class has got many overloaded ‘append’ methods which can be used based on the application need.

#### 11. **setLength(int newLength)**

Sets the length of this String buffer.

```
public class StringBufferFunctionsDemo {

    public static void main(String[] args) {
        //      Examples of Creation of Strings
        StringBuffer strBuf1 = new StringBuffer("Bobby");
        StringBuffer strBuf2 = new StringBuffer(100); //With capacity 100
        StringBuffer strBuf3 = new StringBuffer(); //Default Capacity 16
        System.out.println("strBuf1 : " + strBuf1);
        System.out.println("strBuf1 capacity : " + strBuf1.capacity());
        System.out.println("strBuf2 capacity : " + strBuf2.capacity());
        System.out.println("strBuf3 capacity : " + strBuf3.capacity());
        System.out.println("strBuf1 length : " + strBuf1.length());
        System.out.println("strBuf1 charAt 2 : " + strBuf1.charAt(2));
        //      A StringIndexOutOfBoundsException is thrown if the index is not valid.
        strBuf1.setCharAt(1, 't');
        System.out.println("strBuf1 after setCharAt 1 to t is : "
                           + strBuf1);
        System.out
            .println("strBuf1 toString() is : " + strBuf1.toString());
        strBuf3.append("beginner-java-tutorial");
        System.out.println("strBuf3 when appended with a String : "
                           + strBuf3.toString());
        strBuf3.insert(1, 'c');
        System.out.println("strBuf3 when c is inserted at 1 : "
                           + strBuf3.toString());
        strBuf3.delete(1, 'c');
        System.out.println("strBuf3 when c is deleted at 1 : "
                           + strBuf3.toString());
        strBuf3.reverse();
        System.out.println("Reversed strBuf3 : " + strBuf3);
        strBuf2.setLength(5);
        strBuf2.append("jdbc-tutorial");
        System.out.println("strBuf2 : " + strBuf2);
        //      We can clear a StringBuffer using the following line
        strBuf2.setLength(0);
        System.out.println("strBuf2 when cleared using setLength(0): "
                           + strBuf2);
    }
}
```

```
}
```

***Output***

```
strBuf1 : Bobby
strBuf1 capacity : 21
strBuf2 capacity : 100
strBuf3 capacity : 16
strBuf1 length : 5
strBuf1 charAt 2 : b
strBuf1 after setCharAt 1 to t is : Btbbby
strBuf1 toString() is : Btbbby
strBuf3 when appended with a String : beginner-java-tutorial
strBuf3 when c is inserted at 1 : bbeginner-java-tutorial
strBuf3 when c is deleted at 1 : b
Reversed strBuf3 : b
strBuf2 :
```

## Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to `main()`. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a `String` array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on. For example, the following program displays all of the command-line arguments that it is called with:

```
*****
* File: CommandLine.java
* Purpose: Display all command-line arguments
*****  
  
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "]: " +
                args[i]);
    }
}
```

Try executing this program, as shown here:

`java CommandLine this is a test 100 -1`

When you do, you will see the following output:

`args[0]: this`

`args[1]: is`

`args[2]: a`

args[3]: test  
args[4]: 100  
args[5]: -1

**REMEMBER** All command-line arguments are passed as strings.

## Introduction to Java Classes

- ✓ A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object.
- ✓ Methods are nothing but members of a class that provide a service for an object or perform some business logic.
- ✓ Java fields or member data which used to store information of an object.

A class has the following general syntax:

```
<access modifier>class <classname>
{
<access modifier> member data
<access modifier> member function
}
```

Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified.

## Access Modifiers

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

## **public access modifier**

Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

## **private access modifier**

The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accessed by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

## **protected access modifier**

The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

## **default access modifier**

Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

For better understanding, member level access is formulated as a table:

| <i><b>Access Modifiers</b></i> | <i><b>Same Class</b></i> | <i><b>Same Package</b></i> | <i><b>Subclass</b></i> | <i><b>Other packages</b></i> |
|--------------------------------|--------------------------|----------------------------|------------------------|------------------------------|
| <b>public</b>                  | Y                        | Y                          | Y                      | Y                            |
| <b>protected</b>               | Y                        | Y                          | Y                      | N                            |
| <b>no access modifier</b>      | Y                        | Y                          | N                      | N                            |
| <b>private</b>                 | Y                        | N                          | N                      | N                            |

Below is an example showing the Objects and Classes of the Cube class that defines 3 fields namely length, breadth and height. Also the class contains a member function getVolume().

```
public class Cube {
```

```

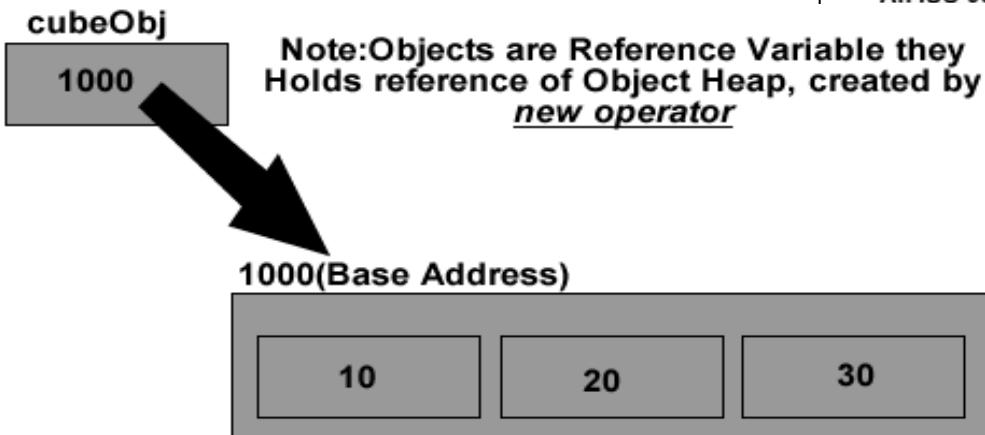
int length;
int breadth;
int height;
public int getVolume() {
    return (length * breadth * height);
}
}

```

## Objects

- ✓ *Objects are instance of classes*
- ✓ *These are basic runtime entities of object oriented programming*
- ✓ *Real-world objects share two characteristics: They all have state and behavior. Cars have state (Colors, Four Tyres, Speed, Engine (petrol/diesel) and behavior (changing gear, applying brakes, increase/decrease speed). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.*
- ✓ *An object is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behavior.*
- ✓ *An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables.*

Note: Instance variables stores the state of the object. Each class would have its own copy of the variable. Every object has a state that is determined by the values stored in the object. An object is said to have changed its state when one or more data values stored in the object have been modified. An object that has the ability to store values is often said to have persistence



Ps-Softech,Gwalior

```
*****
* File: Cube.java
* Purpose: Program to Find out Volume of cube using class & object
*****
import java.util.*;
public class Cube {
    private int length,breadth,height;
    Scanner KB=new Scanner(System.in);
    void getValues()
    {System.out.print("Enter Length:");
        length=KB.nextInt();
        System.out.print("Enter Breadth:");
        breadth=KB.nextInt();
        System.out.print("Enter Height:");
        height=KB.nextInt();
    }
    int getVolume() {
        return (length * breadth * height);
    }
    public static void main(String[] args) {
        Cube cubeObj; // Creates a Cube Reference
        cubeObj = new Cube(); // Creates an Object of Cube
        cubeObj.getValues();
    }
}
```

```
int v=cubeObj.getVolume();
System.out.println("Volume of Cube is : " + v);
}}
```



C:\Windows\system32\cmd.exe

```
Enter Length:10
Enter Breadth:20
Enter Height:30
Volume of Cube is : 6000
Press any key to continue . . .
```

## Java Methods

### Parameterized Member Functions

A parameter or an argument is a value that is passed to a method so that the method can use that value in its processing.

Note: Java uses only pass by value. Primitive data types are passed purely as pass by value this means when we pass primitive datatypes to method it will pass only values to function parameter so any changes made in parameter will not affect the value of actual parameters, whereas for objects a value which is the reference to the object is passed. Hence the whole object is not passed but its reference gets passed. All modifications to the object in the method would modify the object in the Heap.

#### The following rules apply to parameters:

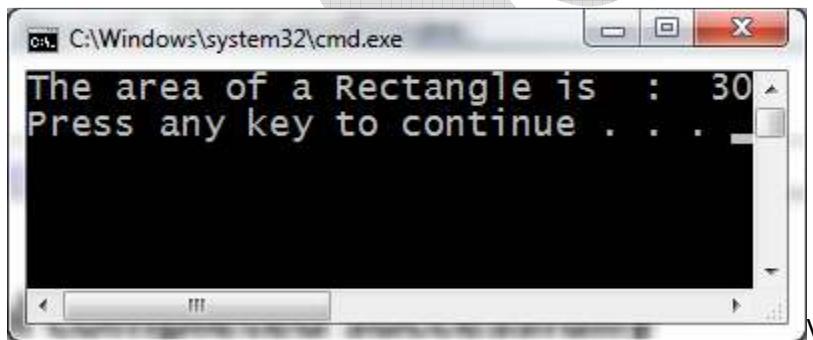
- ✓ You can declare a method without parameters by using an empty pair of parentheses.
- ✓ No duplicate parameters are allowed in a method.
- ✓ The scope of a parameter is the entire body of the method.

- ✓ If a parameter has the same name as a class member, it is the parameter (not the class member) that is referred to within the body of that method. The class member is *hidden* by the parameter.

### ***Passing Primitive Data types(call by value):***

```
*****
* File: Rectangle.java
* Purpose: Program to Find out area of rectangle using parameterized member Function
*****
```

```
class Rectangle{
    int length,breadth;
    void show(int length,int breadth){
        this.length=length;
        this.breadth=breadth;
    }
    int calculate(){
        return(length*breadth);
    }
}
public class UseOfThisOperator{
    public static void main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.show(5,6);
        int area = rectangle.calculate();
        System.out.println("The area of a Rectangle is : " + area); }}
```



### **this object**

In java there's "this" keyword. It can be used inside methods (and constructors).

It returns the reference to the current object.

The keyword **this** is useful when you need to refer to instance of the class from its method. The keyword helps us to avoid name conflicts. As we can see in the program that we have declare the name of instance variable and local variables same. Now to avoid the conflict between them we use **this** keyword. Here, this section provides you an example with the complete code of the program for the illustration of how to what is **this** keyword and how to use it.

In the above example, **this.length** and **this.breadth** refers to the instance variable length and breadth while length and breadth refers to the arguments passed in the method. We have made a program over **this**. After going through it you can better understand.

### **Passing Object as an Arguments(Call By Reference)**

*Object as an argument is use to establish communication between two or more objects of same or different class,i.e. user can easily process data of two same or different object within functions.*

Example: Add Two Time Objects,Distance Objects, Matrix Objects etc.

Note:When an argument is an object the "value" passed is actually a reference (pointer) to that object.

### **Example Object as an Arguments and Return Type**

```
*****
* File: twonum.java
* Purpose: Passing Object as an arguments and returning objects
*****
class twonum
{ private int x,y;
void getvalues(int x,int y)
{ this.x=x;
this.y=y;
}
void putvalues()
{System.out.println("x:"+x+" y:"+y);
}
```

```

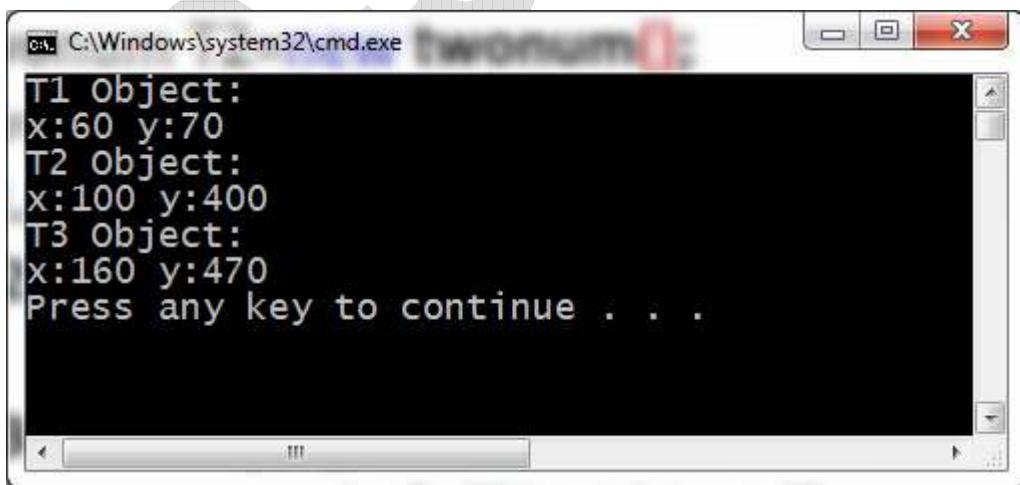
twonum add(twonum A)//A is an Object as arguments
{
    twonum R=new twonum();
    R.x=x+A.x;
    R.y=y+A.y;
    return(R);
}

}
class objet
{ public static void main(String arg[])
{ twonum T1=new twonum();
    twonum T2=new twonum();
    twonum T3;
    T1.getvalues(60,70);
    T2.getvalues(100,400);

    T3=T1.add(T2);//T2 Object Passed its Reference to Object A while Calling Function
    System.out.println("T1 Object:");
    T1.putvalues();
    System.out.println("T2 Object:");

    T2.putvalues();
    System.out.println("T3 Object:");
    T3.putvalues();
}
}

```

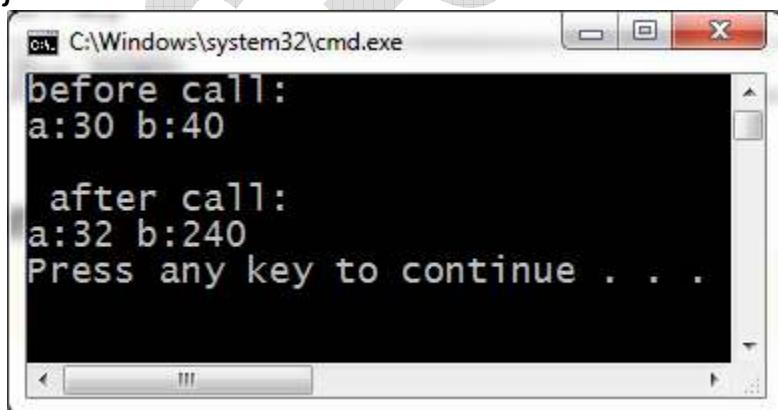


**Example: Objects are passed by reference.**

**REMEMBER** When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

```
*****  
* File: Test.java  
* Purpose: To Test Object Passes as Ref. and Change the Value of Actual Arguments  
*****  
  
class Test {  
    private int a, b;  
    void get(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // pass an object  
    void call(Test T) {  
        T.a=T.a+2;  
        T.b=T.b+200;  
    }  
    void show()  
    {System.out.println("a:"+a+" b:"+b);  
    }  
}
```

```
class CallByRef {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        ob.get(30,40);  
        System.out.println("before call: ");  
        ob.show();  
        ob.call(ob); //ob passed its reference to T  
        System.out.println("\n after call: ");  
        ob.show();  
    }  
}
```



# Method Overloading

*Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. This allows the compiler to match parameters and choose the correct method when a number of choices exist. Changing just the return type is not enough to overload a method, and will be a compile-time error. They must have a different signature. When no method matching the input parameters is found, the compiler attempts to convert the input parameters to types of greater precision. A match may then be found without error. At compile time, the right implementation is chosen based on the signature of the method call.*

Below is an example of a class demonstrating Method Overloading

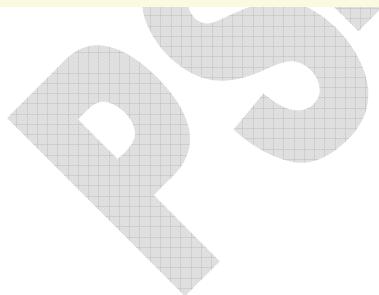
```
*****
```

\* File: MethodOverloadDemo.java

\* Purpose: Method Overloading

\*\*\*\*\*\*/

```
public class MethodOverloadDemo {  
    void sum() { // First Version  
        System.out.println("No parameters");  
    }  
    void sum(int a) { // Second Version  
        System.out.println("One parameter: " + a);  
    }  
    int sum(int a, int b) { // Third Version  
        System.out.println("Two parameters: " + a + ", " + b);  
        return a + b;  
    }  
    double sum(double a, double b) { // Fourth Version  
        System.out.println("Two double parameters: " + a + ", " + b);  
        return a + b;  
    }  
    public static void main(String args[]) {  
        MethodOverloadDemo moDemo = new MethodOverloadDemo();  
        int intResult;  
        double doubleResult;  
        moDemo.sum();  
        System.out.println();  
        moDemo.sum(2);  
        System.out.println();  
        intResult = moDemo.sum(10, 20);  
        System.out.println("Sum is " + intResult);  
        System.out.println();  
        doubleResult = moDemo.sum(1.1, 2.2);  
        System.out.println("Sum is " + doubleResult);  
        System.out.println();  
    }  
}
```





```
C:\Windows\system32\cmd.exe
No parameters
One parameter: 2
Two parameters: 10 , 20
Sum is 30
Two double parameters: 1.1 , 2.2
Sum is 3.3000000000000003
Press any key to continue . . .
```

## Constructors

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.

### Properties of Constructors

- ✓ **Constructor name is class name.** A constructors must have the *same name as the class its in.*
- ✓ **Default constructor.** If you don't define a constructor for a class, a *default parameter less constructor* is automatically created by the compiler. The default constructor calls the default parent constructor (`super()`) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).
- ✓ **Default constructor is created only if there are no constructors.** If you define *any* constructor for your class, no default constructor is automatically created.
- ✓ **Differences between methods and constructors.**
  - There is *no return type* given in a constructor signature (header). The value is this object itself so there is no need to indicate a return value.
  - There is *no return statement* in the body of the constructor.

### Types of Constructor

1) Default Constructor(0 parameter constructor)

2) Parameterized Constructors

3) Overloaded Constructor

```
*****
* File: BoxDemo.java
* Purpose: Here, Box uses a constructor to initialize the dimensions of a box.
*****
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
```

```

height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}

```

When this program is run, it generates the following results:

Constructing Box  
Constructing Box  
Volume is 1000.0  
Volume is 1000.0

### Parameterized Constructors

- ✓ While the **Box()** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions.
- ✓ The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
*****  
* File: BoxDemo.java  
* Purpose: Here, Box uses a parameterized constructor to initialize the dimensions of a box.  
*****/  
  
class Box {  
    double width;  
    double height;  
    double depth;  
    // This is the constructor for Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}  
class BoxDemo{  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
        double vol;  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

The output from this program is shown here:

```
Volume is 3000.0  
Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

Box mybox1 = new Box(10, 20, 15);  
 the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.  
 Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15,  
 respectively.

### Overloaded Constructor

*A class contains more than one constructor with different parameter known as overloaded constructors.*

```
*****  

* File: cons.java  

* Purpose: Overloaded Constructor  

*****
```

```
import java.util.*;  

class person  

{ private String name;  

  private int age;  

  Scanner KB=new Scanner(System.in);  

  person(){  

  person(String name,int age)  

  { this.name=name;  

    this.age=age;  

  }  

  void getperson()  

  {System.out.print("Enter Name:");  

   name=KB.next();  

   System.out.print("Enter Age:");  

   age=KB.nextInt();  

  }  

  void putperson()  

  {System.out.println("Name:"+name+" Age:"+age);  

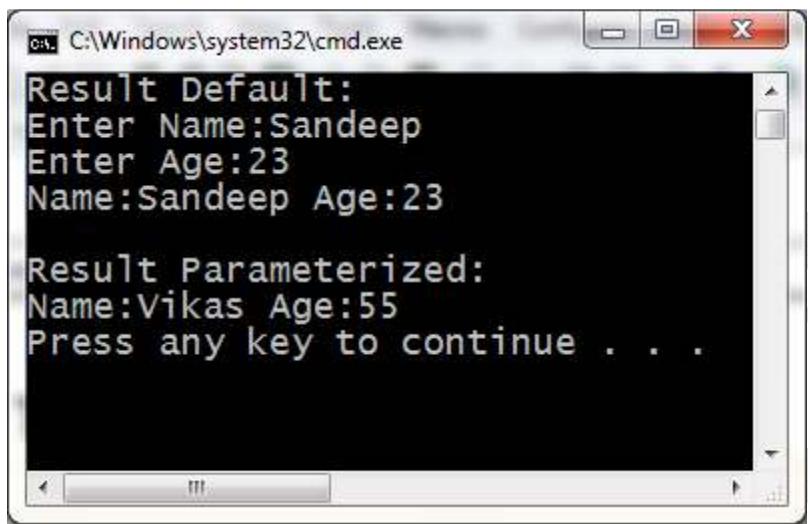
  }  

}  
  

class cons  

{ public static void main(String arg[])
  { person p=new person();//default
    System.out.println("Result Default:");
    p.getperson();
    p.putperson();
    System.out.println("\nResult Parameterized:");
  }
}
```

```
person p1=new person("Vikas",55);//parameterized  
p1.putperson();  
}}
```



## Destroying Objects(Garbage Collection)

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

**Note:** The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects and destroys the objects have no reference.

### ✓ How one can Destroy Objects In Java

When any Instantiated object assign with null or lost their reference(any case) the Java Run Time Environment(JVM) marks the location of object as Garbage Collection(GC), Now it's the responsibility of main thread(Low Priority Thread) to destroy those location which marked with GC.

### ✓ System.gc()

In Java the GC runs automatically, but you can also call it explicitly with [System.gc\(\)](#) and try to force a major garbage collection.

`System.gc()` destroy those objects which marked with GC on user request.

### ✓ The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. Before removing an object from memory **Garbage collection thread invokes finalize () method** of that object and gives an opportunity to perform any sort of cleanup required

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed.

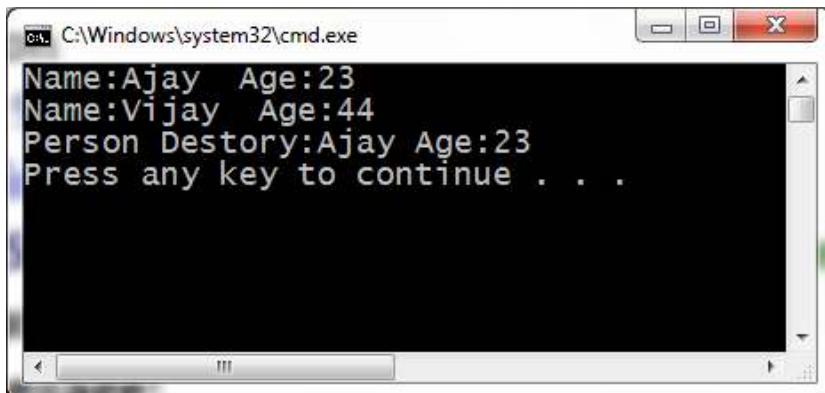
```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. **It is important to understand that finalize( ) method invokes automatically before destroying any objects.**

```
*****
* File: gc.java
* Purpose: Destroying Object at Run Time
*****
```

```
class person
{ private String name;
  private int age;
  person(String name,int age)//Parameterized constructor
  { this.name=name;
    this.age=age;
  }
  void putperson()
  {System.out.println("Name:"+name+" Age:"+age);
   }
  protected void finalize()
  { System.out.println("Person Destory:"+name+" Age:"+age);
  }
}

class gc
{
  public static void main(String arg[])
  { person p1=new person("Ajay",23);
    person p2=new person("Vijay",44);
    p1.putperson();
    p2.putperson();
    p1=null; //object assign with null marked as GC
    System.gc();
  }
}
```



```
C:\Windows\system32\cmd.exe
Name:Ajay Age:23
Name:Vijay Age:44
Person Destory:Ajay Age:23
Press any key to continue . . .
```

## **Static Member**

The static keyword can be used in **3 scenarios**

- 1) **static variables**
- 2) **static methods**
- 3) **static blocks of code.**

### ✓ **static variable:**

A data member of a class can be qualified as static. A static member variable has certain special characteristics. These are:

- It is a variable which **belongs to the class** and not to **object(instance)**
- Static variables are **initialized only once**, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- A static variable can be **accessed directly** by the **class name** and doesn't need any object
- Syntax : <class-name>.<variable-name>
- Since they are associated with the class itself rather than with any class object, they are also known as **class variables**.

### ✓ **static method**

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

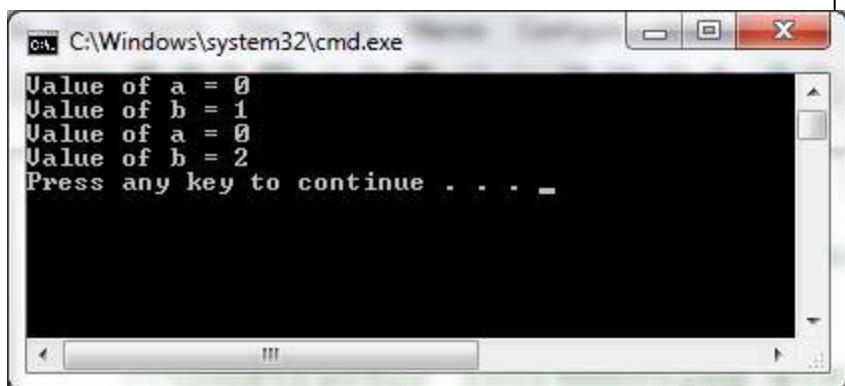
- It is a method which **belongs to the class** and **not to the object**(instance)
- A static method **can access only static data**. It can not access non-static data (instance variables)
- A static method **can call only other static methods** and can not call a non-static method from it.
- A static method **can be accessed directly** by the **class name** and doesn't need any object
- Syntax : <class-name>.<method-name>
- A static method cannot refer to "this" or "super" keywords in anyway

```
*****
* File: Demo.java
* Purpose: Example Static Function and Datamember
*****/
class Student {
    int a; //initialized to zero
    static int b; //initialized to zero only when class is loaded not for each object created.

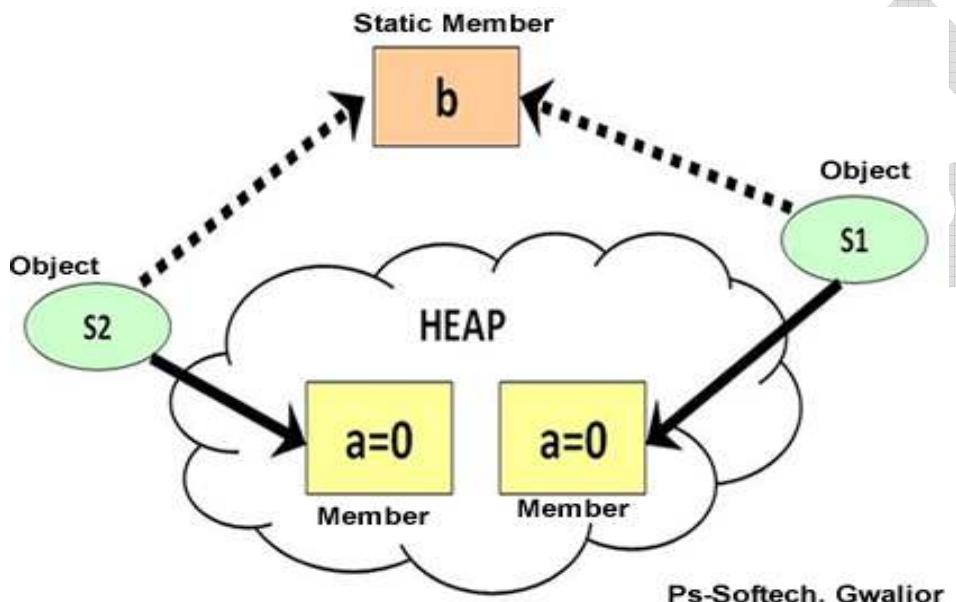
    Student(){
        //Constructor incrementing static variable b
        b++;
    }

    public void showData(){
        System.out.println("Value of a = "+a);
        System.out.println("Value of b = "+b);
    }
}

class Demo{
    public static void main(String args[]){
        Student s1 = new Student();
        s1.showData();
        Student s2 = new Student();
        s2.showData();
    }
}
```



```
C:\Windows\system32\cmd.exe
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
Press any key to continue . . .
```



Ps-Softech, Gwalior

### ✓ static block

The static block, is a block of statement inside a Java class that will be executed when a class is first loaded in to the JVM

```
class Test{
    static {
        //Code goes here
    }
}
```

A **static block helps to initialize the static data members**, just like constructors help to initialize instance members

## Using Wrapper Classes and Boxing and UnBoxing

Wrapper class is a wrapper around a primitive data type. It represents primitive data types in their corresponding class instances e.g. a boolean data type can be represented as a Boolean class instance.

The wrapper classes in the Java API serve two primary purposes:

- To provide a mechanism to “wrap” primitive values in an object so that the primitives can be included in activities reserved for objects, like being added to Collections, or returned from a method with an object return value.
- To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

| Basic Type | Wrapper Class |
|------------|---------------|
| byte       | Byte          |
| short      | Short         |
| int        | Integer       |
| long       | Long          |
| float      | Float         |
| double     | Double        |
| boolean    | Boolean       |
| char       | Character     |

### ✓ Integer Class

#### CONSTRUCTORS

The *Integer* class has two constructors.

One constructor has an *int* as its parameter, the other has a string. We might use them to initialise two *Integer* objects with value five like this:

```
Integer integer1 = new Integer(5);
Integer integer2 = new Integer("5");
```

It would be an error to supply a string argument that could not possibly be converted to an *int*. For example

```
Integer anInteger = new Integer("XX");
```

*would generate a run-time error.*

**Boxing:** Convert any Primitive data into non primitive or Object known as Boxing means Wrapping data into object

**Example:**

```
Integer X=new Integer(7); //Boxing
```

**Unboxing:** convert object data into primitive data.

```
int a=X.intValue(); //Unboxing
```

| Constructors                    |                                                                                                                                                                                                                     |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Integer(int value)</code> | an <i>Integer</i> object that represents the given <i>int</i> value.                                                                                                                                                |
| <code>Integer(String s)</code>  | an <i>Integer</i> object that represents the <i>int</i> value represented by the given string, <i>s</i> . <i>s</i> must represent a valid <i>int</i> (otherwise a <i>NumberFormatException</i> error is generated). |

## METHODS

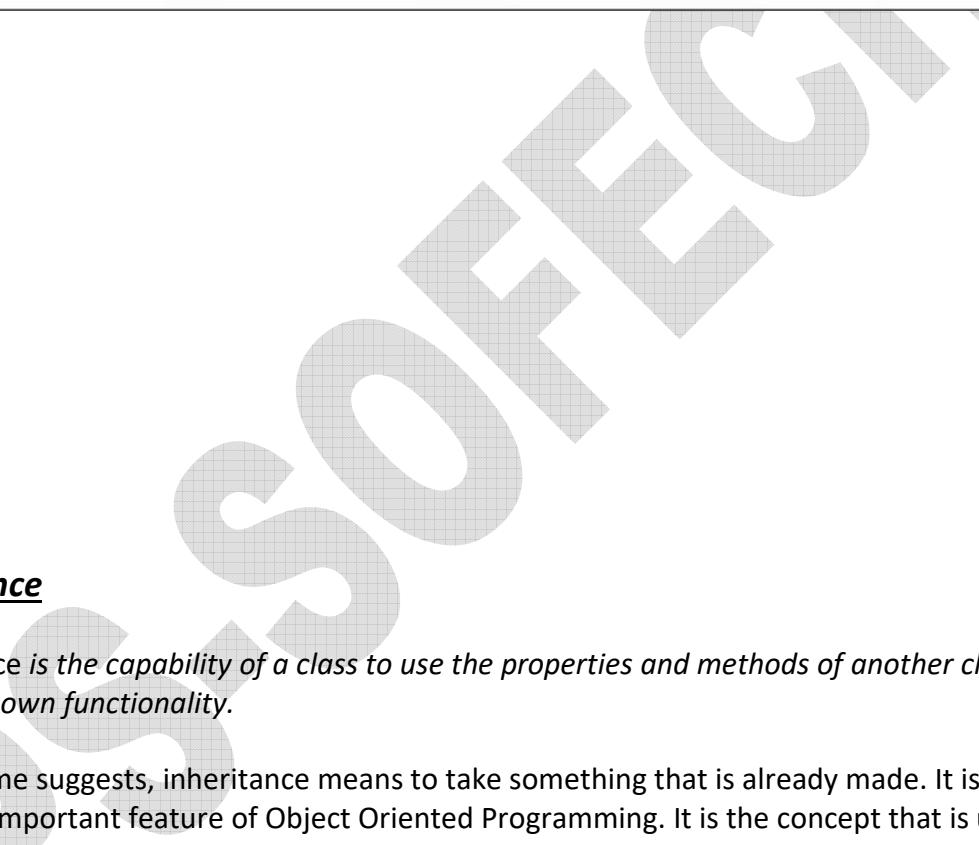
Some useful methods are shown in the following table.

| Methods        |                              |                                                                                                                                                                                                                                                |
|----------------|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int            | compareTo(Integer anInteger) | returns a value less than zero if this Integer's value is less than the given anInteger's value, zero if the two Integer values are the same, returns a value more than zero if this Integer's value is more than the given anInteger's value. |
| double         | doubleValue()                | returns the value of this <i>Integer</i> as a <i>double</i> .                                                                                                                                                                                  |
| boolean        | equals(Object obj)           | returns <i>true</i> if the given object is an <i>Integer</i> object with the same <i>int</i> value as this Integer's <i>int</i> value.                                                                                                         |
| int            | intValue()                   | returns the value of this <i>Integer</i> as an <i>int</i> .                                                                                                                                                                                    |
| static int     | parseInt(String s)           | converts the given string to an <i>int</i> . <i>s</i> must represent a valid <i>int</i> otherwise a <i>NumberFormatException</i> error is generated.                                                                                           |
| String         | toString()                   | returns a string object representing this Integer's value.                                                                                                                                                                                     |
| static String  | toString(int i)              | returns a new string object representing the given <i>int</i> .                                                                                                                                                                                |
| static Integer | valueOf(String s)            | returns a new <i>Integer</i> object initialised to the <i>int</i> value represented by <i>s</i> . <i>s</i> must represent a valid <i>int</i> otherwise a <i>NumberFormatException</i> error is generated.                                      |

**Note:**All Above Function Also Define in Long,Short,Byte,Float and Double Class

```
*****
* File: TestInteger.java
* Purpose: Example Of Integer class and Its Method
*****
public class TestInteger {
public static void main(String[] s)
{
System.out.println("The largest int value is ... " +Integer.MAX_VALUE);
System.out.println("Using the two constructors to " +"create two Integer objects with value 5
... ");
Integer integer1 = new Integer(5);
Integer integer2 = new Integer("5");
System.out.println("Both Integer objects are " +"the same ... " +integer1.equals(integer2));
System.out.println("Converting Strings to ints.");
int i = Integer.parseInt("5") + Integer.parseInt("2");
System.out.println("The result should be 7. " +"It is ... " + i);
System.out.println("Performing arithmetic with " +"Integer objects.");
i = integer1.intValue() + integer2.intValue();
System.out.println("The result should be 10. " +"It is .. " + i);
}
}
```

}



```
C:\Windows\system32\cmd.exe
The largest int value is ... 2147483647
Using the two constructors to create two Integer objects with value 5 ...
Both Integer objects are the same ... true
Converting Strings to ints.
The result should be 7. It is ... 7
Performing arithmetic with Integer objects.
The result should be 10. It is .. 10
Press any key to continue . . .
```

## Inheritance

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality.

As the name suggests, inheritance means to take something that is already made. It is one of the most important feature of Object Oriented Programming. It is the concept that is used for reusability purpose. **Inheritance is the mechanism through which we can derived classes from other classes. The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class. To derive a class in java the keyword extends is used.** To clearly understand the concept of inheritance you must go through the following example.

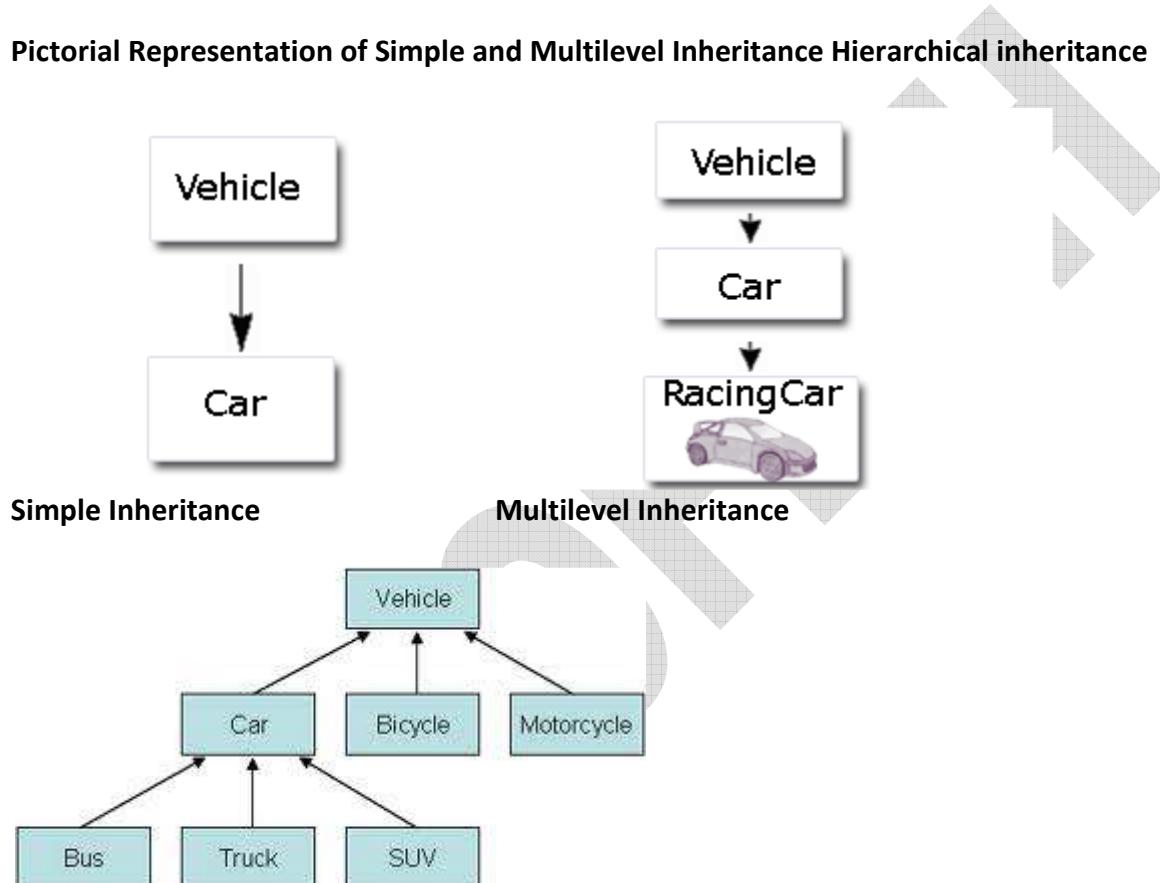
The concept of inheritance is used to make the things from general to more specific e.g. When we hear the word vehicle then we got an image in our mind that it moves from one place to another place it is used for traveling or carrying goods but the word vehicle does not specify whether it is two or three or four wheeler because it is a general word. But the word car makes a more specific image in mind than vehicle, that the car has four wheels . It concludes from the

example that car is a specific word and vehicle is the general word. If we think technically to this example then vehicle is the super class (or base class or parent class) and car is the subclass or child class because every car has the features of it's parent (in this case vehicle) class.

The following kinds of inheritance are there in java.

- Simple Inheritance
- Multilevel Inheritance

#### Pictorial Representation of Simple and Multilevel Inheritance Hierarchical inheritance



The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
// body of class
}
```

- ✓ **Private:** Members which declare private within the class are totally hidden and cannot access by subclass directly, only the member function which declare within the class can access it(public or private);
- ✓ **Public:** Members which declare public within the class can access by derive class (sub class) directly.

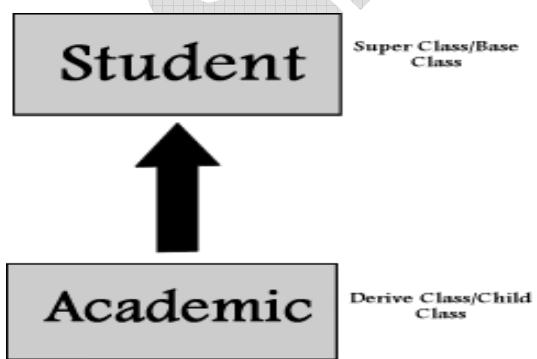
- ✓ **Protected:** Members Declare as protected is access by the class in which they declare as protected and the subclass directly, it cannot access by the member function outside these two classes.

For better understanding, member level access is formulated as a table:

| <i>Access Modifiers</i>   | <i>Same Class</i> | <i>Same Package</i> | <i>Subclass</i> | <i>Other packages</i> |
|---------------------------|-------------------|---------------------|-----------------|-----------------------|
| <b>Public</b>             | Y                 | Y                   | Y               | Y                     |
| <b>Protected</b>          | Y                 | Y                   | Y               | N                     |
| <b>no access modifier</b> | Y                 | Y                   | N               | N                     |
| <b>Private</b>            | Y                 | N                   | N               | N                     |

### Single Inheritance

when a single class is being inherited by a class, it is called single or simple inheritance.



**Ps-Softech Gwalior**

\*\*\*\*\*

\* File: singlein.java

\* Purpose:Single Inheritance Example Students class extends by Academic class

```
import java.util.*;
```

```
class student
```

```
{ private int roll;
```

```
private String sname;
```

```
Scanner X=new Scanner(System.in);
```

```
void getStudent()
```

```
{ System.out.print("Enter Roll No:");
```

```
roll=X.nextInt();
```

```
System.out.print("Enter Name:");
```

```
sname=X.next();
```

```
}
```

```
void putStudent()
```

```
{ System.out.print("Roll No:"+roll);
```

```
System.out.print(" Name:"+sname);
```

```
}
```

```
}
```

```
class academic extends student
```

```
{ private int p,c,m;
```

```
void getAcademic()
```

```
{ getStudent();
```

```
System.out.print("Enter Physics Marks:");
```

```
p=X.nextInt();
```

```
System.out.print("Enter Chemistry Marks:");
```

```
c=X.nextInt();
```

```
System.out.print("Enter Maths Marks:");
```

```
m=X.nextInt();
```

```
}
```

```
void putAcademic()
```

```
{ putStudent();
```

```
System.out.println(" Physics:"+p+" Chemistry:"+c+" Maths:"+m);
```

```
}}
```

```
class singlein
```

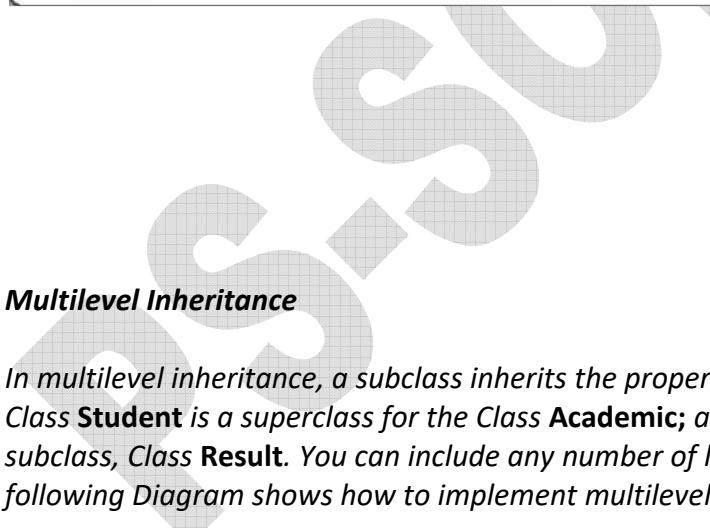
```
{ public static void main(String arg[])
```

```
{ academic s=new academic();
```

```
//derive class constructor always invokes the base class default constructor implicitly
```

```
s.getAcademic();
```

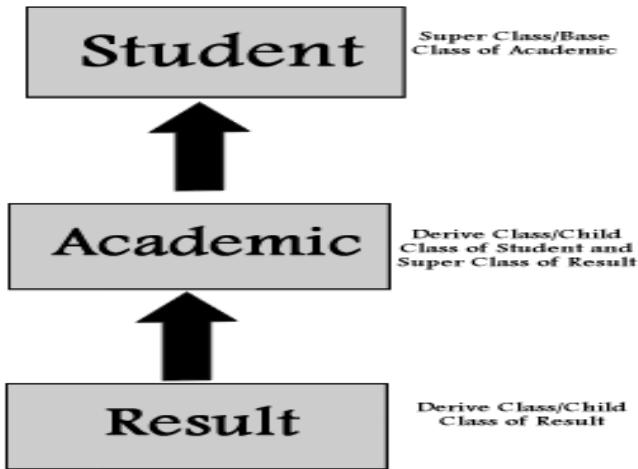
```
s.putAcademic();  
}  
}
```



```
C:\Windows\system32\cmd.exe  
Enter Roll No:100  
Enter Name:Ajay  
Enter Physics Marks:90  
Enter Chemistry Marks:98  
Enter Maths Marks:99  
Roll No:100 Name:Ajay Physics:90 Chemistry:98 Maths:99  
Press any key to continue . . .
```

### Multilevel Inheritance

In multilevel inheritance, a subclass inherits the properties of another subclass. For example, Class **Student** is a superclass for the Class **Academic**; and Class **Academic** is a superclass for the subclass, Class **Result**. You can include any number of levels in multilevel inheritance. The following Diagram shows how to implement multilevel inheritance:



```

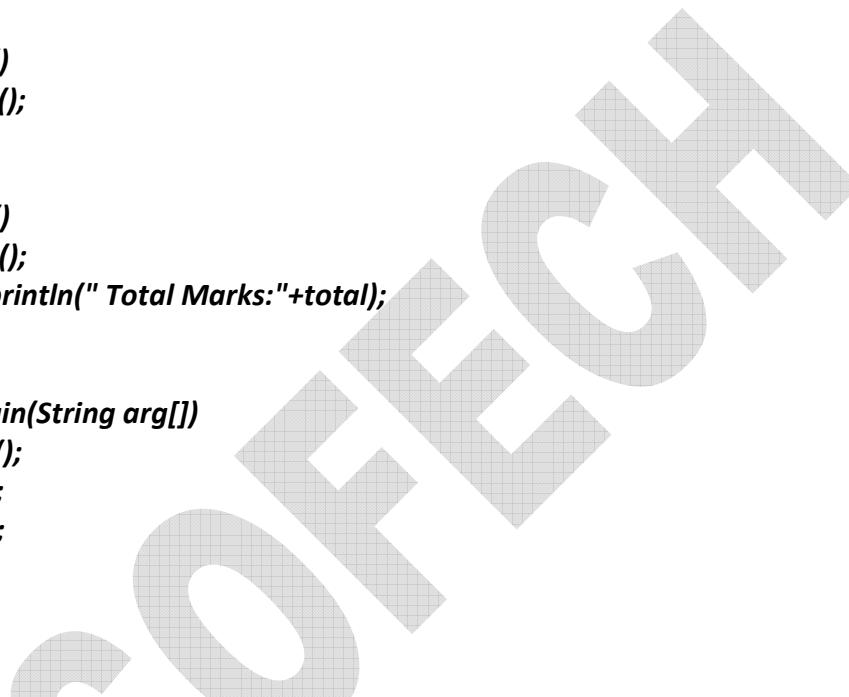
 ****
 * File: multilevel.java
 * Purpose: Example MultiLevel Inheritance(protected members)
 ****

import java.util.*;
class student
{ private int roll;
  private String sname;
  Scanner X=new Scanner(System.in);
  void getStudent()
  { System.out.print("Enter Roll No:");
    roll=X.nextInt();
    System.out.print("Enter Name:");
    sname=X.next();
  }
  void putStudent()
  { System.out.print("Roll No:"+roll);
    System.out.print(" Name:"+sname);
  }
}
class academic extends student
{ protected int p,c,m;
  void getAcademic()
  { getStudent();
    System.out.print("Enter Physics Marks:");
    p=X.nextInt();
    System.out.print("Enter Chemistry Marks:");
    c=X.nextInt();
    System.out.print("Enter Maths Marks:");
  }
}
  
```

```
m=X.nextInt();
        }
    void putAcademic()
    { putStudent();
        System.out.println(" Physics:"+p+" Chemistry:"+c+" Maths:"+m);
    }

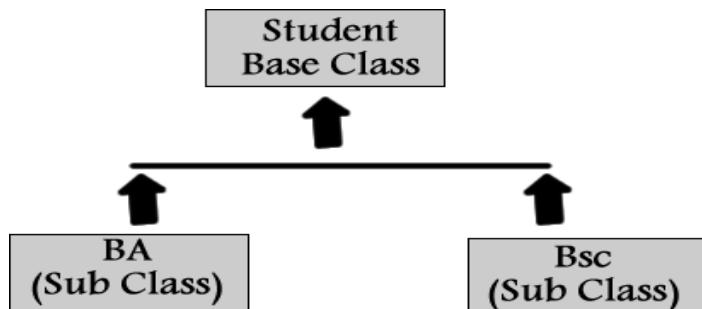
class result extends academic
{ private int total;
    void getResult()
    { getAcademic();
        total=p+c+m;
    }
    void putResult()
    { putAcademic();
        System.out.println(" Total Marks:"+total);
    }
}
class multilevel
{ public static void main(String arg[])
{ result s=new result();
    s.getResult();
    s.putResult();
}}

```



```
C:\Windows\system32\cmd.exe
Enter Roll No:100
Enter Name:Ajay
Enter Physics Marks:90
Enter Chemistry Marks:98
Enter Maths Marks:99
Roll No:100 Name:Ajay Physics:90 Chemistry:98 Maths:99
Total Marks:287
Press any key to continue . . . -
```

In case of hierarchical inheritance we derive more than one subclass from a single super class. In following diagram Student is super class have two subclass BA & Bsc



Ps-Softech,Gwalior

```

 ****
 * File: multilevel.java
 * Purpose: Example Tree Type Inheritance
 ****
import java.util.*;
class student
{ private int roll;
  private String sname;
  Scanner X=new Scanner(System.in);
  void getStudent()
  { System.out.print("Enter Roll No:");
    roll=X.nextInt();
    System.out.print("Enter Name:");
    sname=X.next();
  }
  void putStudent()
  { System.out.print("Roll No:"+roll);
    System.out.print(" Name:"+sname);
  }
}
class bsc extends student
{ protected int p,c,m;
  void getBsc()
  { getStudent();
    System.out.print("Enter Physics Marks:");
    p=X.nextInt();
    System.out.print("Enter Chemistry Marks:");
    c=X.nextInt();
    System.out.print("Enter Maths Marks:");
    m=X.nextInt();
  }
}
  
```

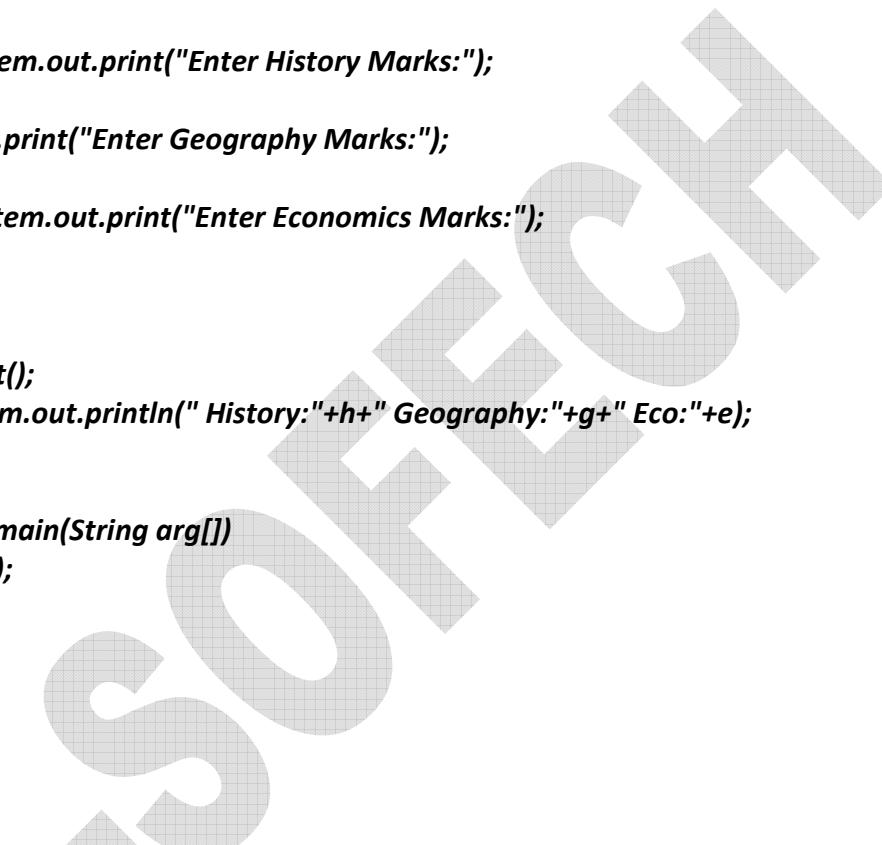
```

void putBsc()
{ putStudent();
    System.out.println(" Physics:"+p+" Chemistry:"+c+" Maths:"+m);
}

class ba extends student
{ protected int h,g,e;
    void getBa()
{ getStudent();
    System.out.print("Enter History Marks:");
h=X.nextInt();
    System.out.print("Enter Geography Marks:");
g=X.nextInt();
    System.out.print("Enter Economics Marks:");
e=X.nextInt();
}
    void putBa()
{ putStudent();
    System.out.println(" History:"+h+" Geography:"+g+" Eco:"+e);
}

class tree
{ public static void main(String arg[])
{ bsc s1=new bsc();
s1.getBsc();
s1.putBsc();
ba s2=new ba();
s2.getBa();
s2.putBa();
}}

```



C:\Windows\system32\cmd.exe

```

Enter Roll No:100
Enter Name:Ajay
Enter Physics Marks:90
Enter Chemistry Marks:89
Enter Maths Marks:90
Roll No:100 Name:Ajay Physics:90 Chemistry:89 Maths:90
Enter Roll No:200
Enter Name:Vikas
Enter History Marks:45
Enter Geography Marks:67
Enter Economics Marks:89
Roll No:200 Name:Vikas History:45 Geography:67 Eco:89
Press any key to continue . .

```

## Constructor in Inheritance(Super() KeyWord)

- ✓ A derived class inherits the members of its base class. Therefore, when a derived class object is instantiated, its base class members must be initialized in addition to its own members means it's the responsibility of derive class constructor(any) to invoke the base class constructor. By default derive class constructor any invokes the base class default constructor implicitly.
- ✓ If base class contains parameterized constructor then it must be invoke with the help of key word super(), this keyword invokes the base class constructor according to parameter specified in base class constructor. Super() keyword must be the first statement in derive class constructor otherwise it will generate error message.

```
*****
* File:cons.java
* Purpose Constructor in Inheritance
*****
```

```
class A
{
    A()//default constructor
    {System.out.println("Base Version");
    }
    A(int x)//Parameterized constructor
    {System.out.println("Base Version:"+x);
    }
}

class B extends A
{
    B()//invokes the base class default constructor implicitly
    { System.out.println("Derive Version 1" );
    }

    B(int x)
    { super(x); //invokes the base class parameterized constructor
    System.out.println("Derive Version 2" );
    }

}

class cons
{
public static void main(String arg[])
{
    B b1=new B(); //invoke derive class default constructor
}
```

```
B b2=new B(33);//invoke derive class Parameterized constructor
} }
```

```
C:\Windows\system32\cmd.exe
Base Version
Derive Version 1
Base Version:33
Derive Version 2
Press any key to continue . . .
```

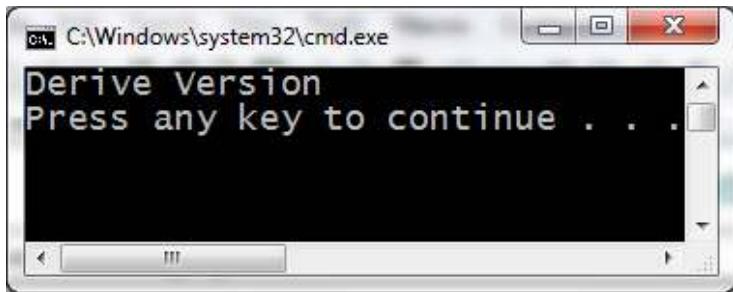
### Method Overriding (Super Keyword)

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

If you wish to access the superclass version of an overridden function, you can do so by using **super**.

```
*****
* File:overriding.java
* Purpose: Function Overriding in Inheritance
*****
```

```
class A
{
    void show()
    {System.out.println("Base Version");
     }
}
class B extends A
{ void show() //show override base version
{
    System.out.println("Derive Version" );
}
}
class overriding
{ public static void main(String arg[])
{ B b=new B();
  b.show();
}
```



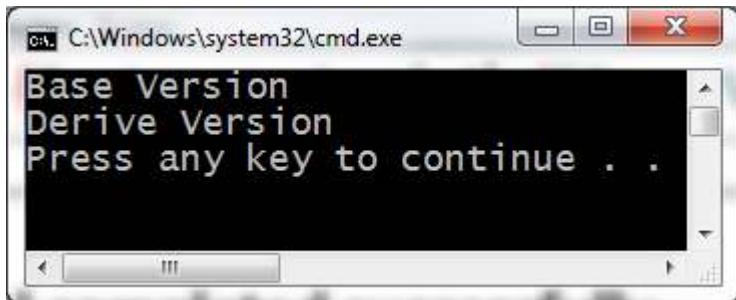
When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden function, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
*****
* File:overriding.java
* Purpose Function Overriding in Inheritance,Using Super keyword to call override member through
derive class
*****
```

```
class A
{
    void show()
    {System.out.println("Base Version");
     }
}

class B extends A
{ void show() //show override base version
  {super.show(); //invokes base version show
   System.out.println("Derive Version" );
   }
}

class overriding
{ public static void main(String arg[])
  { B b=new B();
    b.show();
    }
}
```



Here, **super.show( )** calls the superclass version of **show( )**. Method overriding occurs *only* when the names and the type signatures of the two methods are identical. **If they are not, then the two methods are simply overloaded.**

**Note:** In above example one can't achieve polymorphism because show function invoke by two different method first via keyword super and then through derive class object D.

### Dynamic Method Dispatch (Run Time Polymorphism)

- ✓ In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- ✓ Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.
- ✓ To achieve dynamic polymorphism following rules must be obeyed.
  1. There must be method overriding.
  2. Subclass object must be assigned to a super class object.

```
*****
* File:overriding.java
* Purpose: Dynamic Method Dispatch,Runtime Polymorphism
*****
class A {
void callme() {
System.out.println("Inside A's callme method");
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
```

```

    }
    class C extends A {
        // override callme()
        void callme() {
            System.out.println("Inside C's callme method");
        }
    }
    class Dispatch {
        public static void main(String args[]) {
            A a = new A(); // object of type A
            B b = new B(); // object of type B
            C c = new C(); // object of type C
            A r; // obtain a reference of type A
            r = a; // r refers to an A object
            r.callme(); // calls A's version of callme
            r = b; // r refers to a B object
            r.callme(); // calls B's version of callme
            r = c; // r refers to a C object
            r.callme(); // calls C's version of callme }
        }

```

The output from the program is shown here:

**Inside A's callme method**

**Inside B's callme method**

**Inside C's callme method**

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A's callme( )** method.

**Advantage:** Dynamic, run-time polymorphism is one of the most powerful mechanisms provide reusability of code and robustness, so that user can save lot of memory space at runtime.

**Disadvantage:** Late Binding

**NOTE** Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

## Abstract Classes

### [Run Time Polymorphism]

- ✓ An abstract class is a class that is declared by using the **abstract** keyword. It may or may not have abstract methods. Abstract classes cannot be instantiated, but they can be extended into sub-classes.
- ✓ To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.

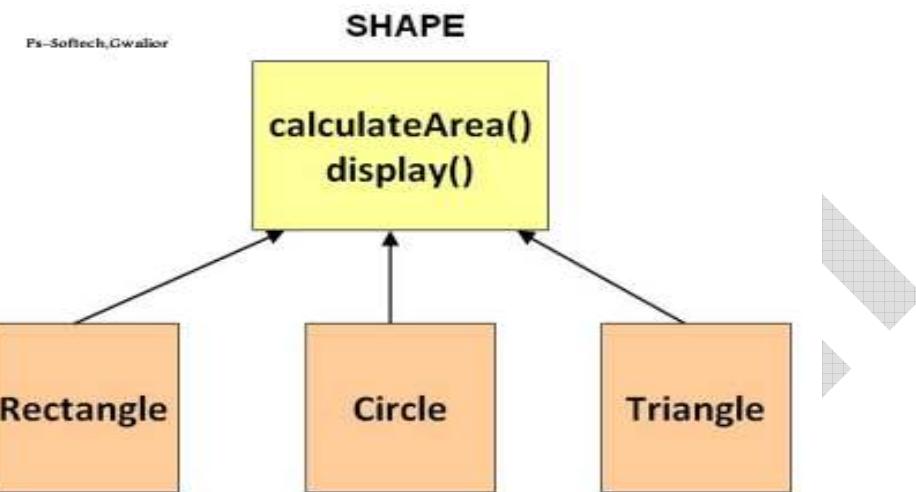
### Points of abstract class :

1. Abstract class contains abstract methods.
2. Program can't instantiate an abstract class.
3. Abstract classes contain mixture of non-abstract and abstract methods.
4. If any class contains abstract methods then it must implements all the abstract methods of the abstract class.

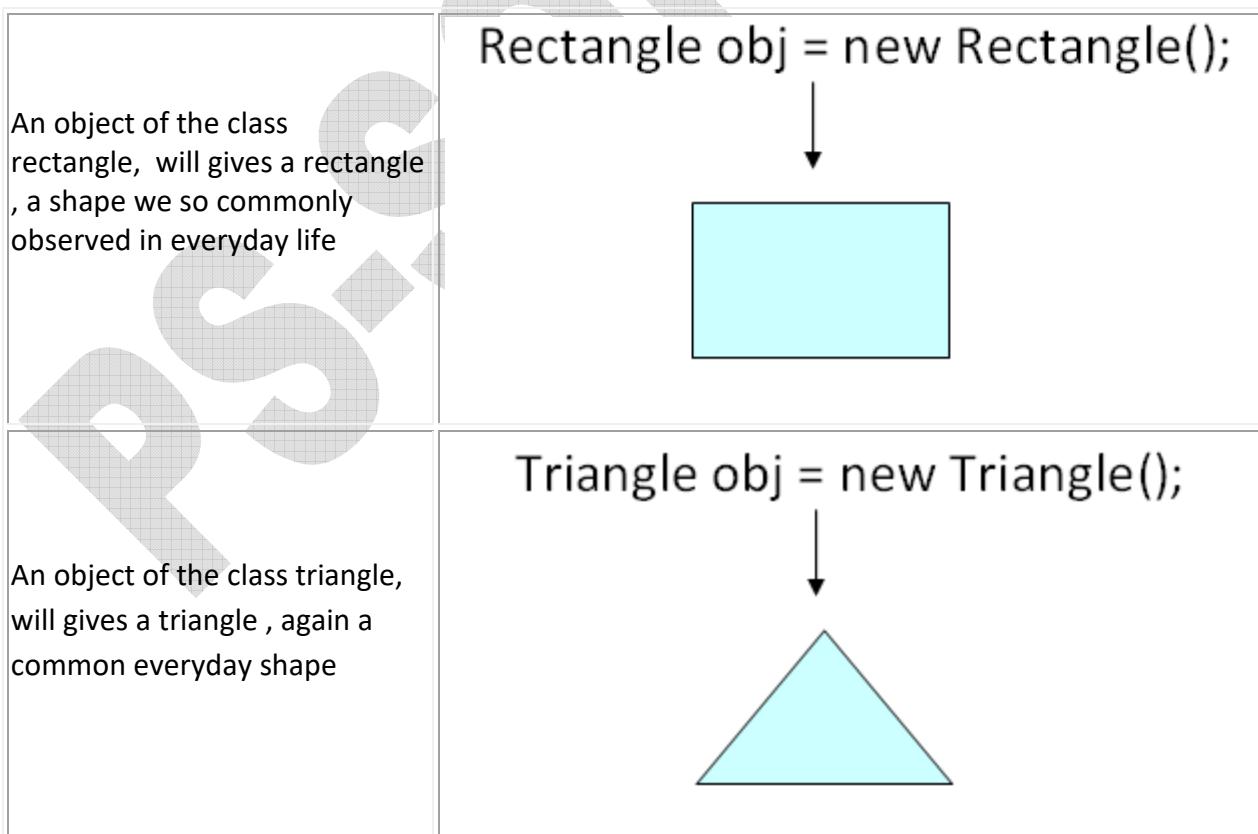
## Abstract Methods:

- ✓ This means that it will not have an implementation. It also means that it MUST appear inside an abstract class.
- ✓ The main property of abstract method is that they must be implemented in subclass or derive class to achieve run time polymorphism.

Consider the following class hierarchy consisting of a Shape class which is inherited by three classes Rectangle , Circle and Triangle. The Shape class is created to save on common attributes and methods shared by the three classes Rectangle , Circle and Triangle. calculateArea() is one such method shared by all 3 child classes and present in Shape class.



Now, assume you write code to create objects for the classes depicted above. Lets observe how these **objects will look in a practical world**.



But what would an object of Class Shape look like in a practical world ??

Shape obj = new Shape();



???

If you observe the Shape class serves in **our goal of achieving inheritance and polymorphism**. But it was not built to be instantiated. Such classes can be labeled **Abstract**. An abstract class can not be instantiated.

**Ex.**

```
abstract class Shape{  
// code  
}
```

It is possible that you DO NOT label Shape class as Abstract and then instantiate it. But such object will have no use in your code and will open a room for potential errors. Hence this is not desirable.

### abstract methods

As we all know, the formula for calculation area for rectangle , circle & triangle is different. The calculateArea() method will have to be overridden by the inheriting classes. It makes so sense defining it in the Shape class, **but we need to make sure that all the inheriting classes do have the method.**

Such methods can be labeled **abstract**.

Ex

```
abstract public void calculateArea();
```

For an **abstract method**, no implementation for is required. Only the signature of the method is defined.

### **Abstract Class & Method Important Points**

**An abstract class may also have concrete (complete) methods.**

- For design purpose, a class can be declared abstract even if it does not contain any abstract methods
- Reference of an abstract class can point to objects of its sub-classes thereby achieving run-time polymorphism Ex: `Shape obj = new Rectangle();`
- A class must be compulsorily labeled abstract , if it has one or more abstract methods.

```
*****  
* File: AbstractAreas.java  
* Purpose: Dynamic Method Dispatch,Runtime Polymorphism using abstract keywords  
*****
```

```
// Using abstract methods and classes.  
abstract class Shape{  
    double dim1;  
    double dim2;  
    Shape(double a, double b) {  
        dim1 = a;  
        dim2 = b;  
    }  
    // area is now an abstract method  
    abstract void calculateArea();  
    abstract void display();  
}
```

```
class Rectangle extends Shape {  
    private double area;  
    Rectangle(double a, double b) {  
        super(a, b);  
    }  
    // override area for rectangle
```

```

void calculateArea() {
System.out.println("Inside Area for Rectangle.");
area= dim1 * dim2;
}
void display()
{System.out.println("Area Rectangle is:"+area);}
}
class Triangle extends Shape {
    private double area;
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
void calculateArea() {
System.out.println("Inside Area for Triangle.");
area=dim1 * dim2 / 2;
}
void display()
{System.out.println("Area Tringle is:"+area);}
}

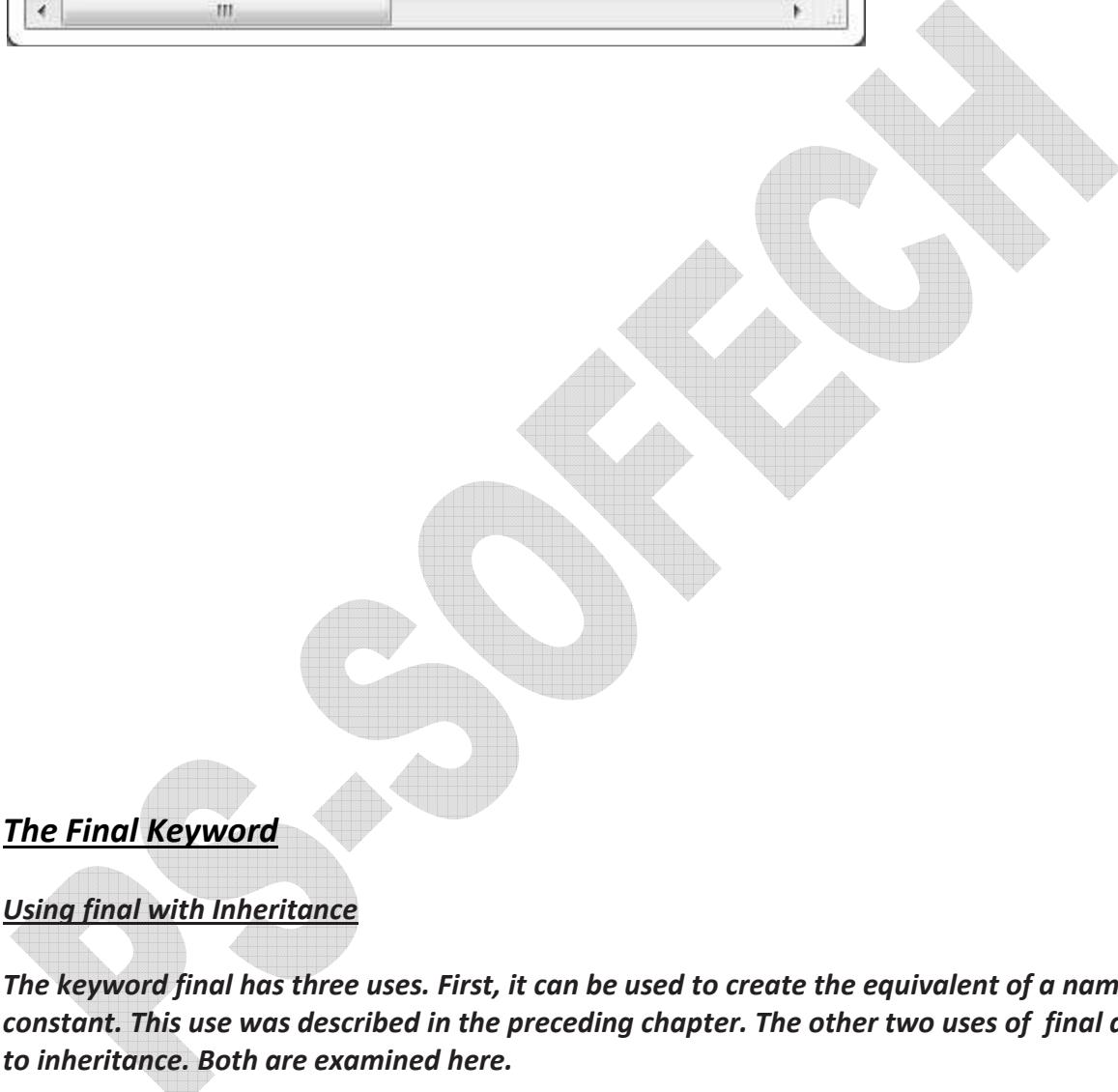
```

```

class AbstractAreas {
public static void main(String args[]) {
// Shape f = new Shape(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Shape SHP; // this is OK, no object is created
//Base Pointer Holds Reference of Rectangle
SHP = r;
SHP.calculateArea();
SHP.display();
//Base Pointer Holds Reference of Tringle

SHP = t;
SHP.calculateArea();
SHP.display();
}
}

```



```
C:\Windows\system32\cmd.exe
Inside Area for Rectangle.
Area Rectangle is:45.0
Inside Area for Triangle.
Area Tringle is:40.0
Press any key to continue . . .
```

## The Final Keyword

### Using final with Inheritance

*The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.*

### Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```

class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}}

```

*Because **meth()** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.*

### **Final Data Member**

The field declared as final behaves like constant. Means once it is declared, it can't be changed. Before compiling, only once it can be set; after that you can't change its value. Attempt to change in its value lead to exception or compile time error.

A variable that is declared as final and not initialized is called a blank final variable. A blank final variable forces the constructors to initialize it.

**Example :**

```

public class FinalTest{
public final int a; // It produces an error "The blank final field a may not have been
initialized".
}

```

- ✓ A blank final variable forces the constructors to initialize it.

```

public class FinalTest{
public final int a;

public TestFinal(){
a=10;
}
}

```

- ✓ A java variable can be declared using the keyword final. Then the final variable can be assigned only once.

```
public final int a =10;
```

```
public static void main(String[] args) {  
TestFinal test = new TestFinal();  
test.a=20;  
  
//Error : The final field TestFinal.a cannot be assigned  
}
```



### Interfaces[Run time polymorphism]

*In java programming language interface is nothing but the collection of methods with empty implementations and constants variables ( variables with static and final declarations ). All the methods in an interface are "public and abstract" by default. Since interfaces are abstract in nature so they can not be directly instantiated. To define the methods of an interface the keyword "implements" is used. Interfaces are similar to abstract classes but the major difference between these two is that interface have all the methods abstract while in case of abstract classes must have at least one abstract method.*

**Points to note:**

- The class which implements the interface needs to provide functionality for the methods declared in the interface
- All methods in an interface are implicitly *public* and *abstract*
- An interface **cannot be instantiated**
- An interface reference can point to objects of its implementing classes
- An interface **can extend** from one or **many interfaces**. A class can extend only one class but implement any number of interfaces

**Syntax**

```
[modifiers] interface InterfaceName {
```

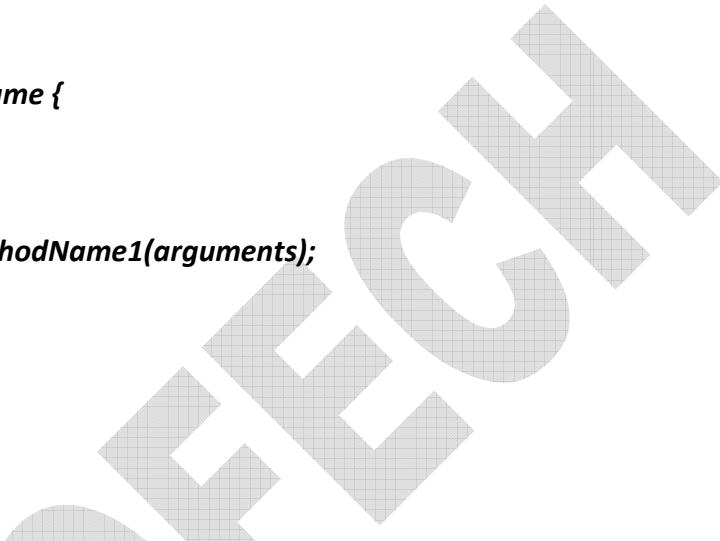
*// declaring methods*

```
[public abstract] returnType methodName1(arguments);
```

*// defining constants*

```
[public static final]
type propertyName = value;
}
```

Ps-Softech Gwalior



```
Sphere Interface
float pi=3.14f; //Final
void getradius(float r); //abstract & public
void show();
```

Class Volume  
implements Sphere

Class Area implements  
Sphere

```
*****
* File: exinter.java
* Purpose: Dynamic Method Dispatch, Runtime Polymorphism using Interface
*****
interface sphere
{ float pi=3.14f;
  void getradius(float r); //abstract & public
  void show();
}
class area implements sphere
{ private float radius,a;
```

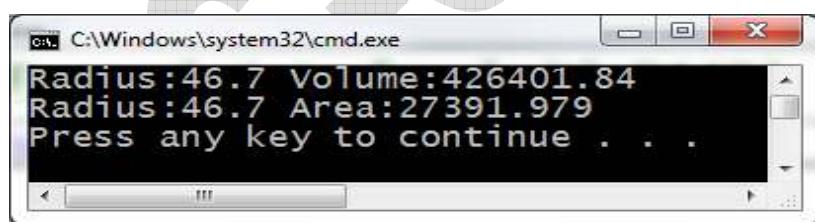
```

public void getradius(float radius)
{ this.radius=radius;
  a=4*pi*radius*radius;
}
public void show()
{System.out.println("Radius:"+radius+" Area:"+a);
}
}

class volume implements sphere
{
  private float radius,v;
  public void getradius(float radius)
  { this.radius=radius;
    v=4f/3f*pi*radius*radius*radius;
  }
  public void show()
  {System.out.println("Radius:"+radius+" Volume:"+v);
  }
}

class exinter
{
  public static void main(String arg[])
  { area A=new area();
    volume V=new volume();
    sphere S;
    S=V;
    S.getradius(46.7f);
    S.show();
    S=A;
    S.getradius(46.7f);
    S.show(); } }

```



### Difference between Interfaces and abstract classes

Some important difference between Interface and abstract classes are given here

| Features | Interface                                        | Abstract Class                                                             |
|----------|--------------------------------------------------|----------------------------------------------------------------------------|
| Methods  | An interface contains all the methods with empty | An abstract class must have at least one method with empty implementation. |

|                             |                                                                                                                     |                                                                                                                             |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
|                             | implementation.                                                                                                     |                                                                                                                             |
| <b>Variables</b>            | The variables in interfaces are final and static.                                                                   | Abstract classes may contain both instance as well as static variables.                                                     |
| <b>Multiple Inheritance</b> | In java multiple inheritance is achieved by using the interface (by implementing more than one interface at a time) | Abstract classes does not provide this functionality.                                                                       |
| <b>Additional Functions</b> | If we add a method to an interface then we will have to implement this interface by any class..                     | In Abstract classes we can add a method with default implementation and then we can use it by extending the abstract class. |

## Packages

### **What Are Packages?**

*Programmers can define their own packages to bundle group of classes/interfaces etc. It is a good practice to group related classes implemented by you so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.*

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

**Some of the existing packages in Java are::**

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

### **Advantages**

- To bundle classes and interfaces
- To know where programmers to find types that can provide related type functions.
- The Classes of one **Package** are isolated from the classes of another **Package**
- The names of your types won't conflict with the type names in other **Packages** because the **Package** creates a new **Namespace**.
- **Packages** provide reusability of code
- We can create our own **Package** or extend already available **Package**

### **Steps to create a Java package:**

1. Come up with a package name
2. Pick up a base directory
3. Make a subdirectory from the base directory that matches your package name.
4. Place your source files into the package subdirectory.
5. Use the package statement in each source file.
6. Compile your source files from the base directory.
7. Run your program from the base directory.

### **Creating a Java Package**

The first step is (rather obviously) to create the directory for the package. This can go in one of two places:

- **in the same folder as the application that will be using it**

- *in a location listed in the system CLASSPATH environmental variable*

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

**Syntax:-**

***package <package\_name>;***

The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

**Example:**

For example, say that we put the files TestA.java and TestB.java into the directory mypack, which is a subdirectory of myApps. So on a MS Windows platform the file path looks like

c:\myApps\mypack\TestA.java

and

c:\myApps\mypack\TestB.java.

At the top of each file we put the statement

**package mypack;**

as shown in the following code:

| <u>myApps/mypack/TestA.java</u>                                                                                                                                                       | <u>myApps/mypack/TestB.java</u>                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>package mypack;  public class TestA {     private int i;     public TestA (int arg1) {         i = arg1;     }      public void show()     {System.out.println(i);     } }</pre> | <pre>package mypack;  public class TestB {     private double x;     public TestB (double y) {         x = y;     }      public void show()     {System.out.println(x);     } }</pre> |

We can illustrate how to use a package with TestAB.java which we put in the next directory above that of the package mypack.

If you working in current folder then there is no need to set the classpath.

But when you are importing packages from other drive or from other folder then it is necessary to set the Classpath.

Set classpath=c:\myApps/mypack;

#### Program to import packages

```
Import mypack.*;
Class testpack
{ public static void main(String arg[])
{ TestA A=new TestA(34);
A.show();
TestB B=new TestB(45.6);
B.show();
}}
```

#### Output

34

456.0

## Exception Handling

*Exception*, that means exceptional errors. Actually *exceptions* are used for handling errors in programs that occurs during the program execution. During the program execution if any error occurs and you want to print your own message or the system message about the error then you write the part of the program which generate the error in the try{} block and catch the errors using catch() block. Exception turns the direction of normal flow of the program control and send to the related catch() block. Error that occurs during the program execution generate a specific object which has the information about the errors occurred in the program.

## Exceptions

- *Errors can be broadly categorized into two groups on the basis of whether the compiler is able to handle the error or not, such as compile time errors and runtime errors.*
- *An exception is a run-time error that can be defined as an abnormal event that occurs during the execution of a program and disrupts the normal flow of instructions.*
- *In Java, the Throwable class is the superclass of all the exception classes. It is the class at the top position in the exception class hierarchy. The Exception class and the Error class are two direct subclasses of the Throwable class.*
- *The built-in exceptions in Java are divided into two types on the basis of the conditions where the exception is raised:*
  - *Checked Exceptions or Compiler-enforced Exceptions*
  - *Unchecked exceptions or Runtime Exceptions*
- *You can implement exception-handling in your program by using the following keywords:*
  - *try*
  - *catch*
  - *throw*
  - *throws*
  - *finally*
- *You use multiple catch blocks to throw more than one type of exception.*
- *The finally clause is used to execute the statements that need to be executed whether or not an exception has been thrown.*
- *The throw statement causes termination of the normal flow of control of the Java code and stops the execution of subsequent statements after the throw statement.*
- *The throws clause is used by a method to specify the types of exceptions the method throws.*
- *You can create your own exception classes to handle the situations specific to an application.*

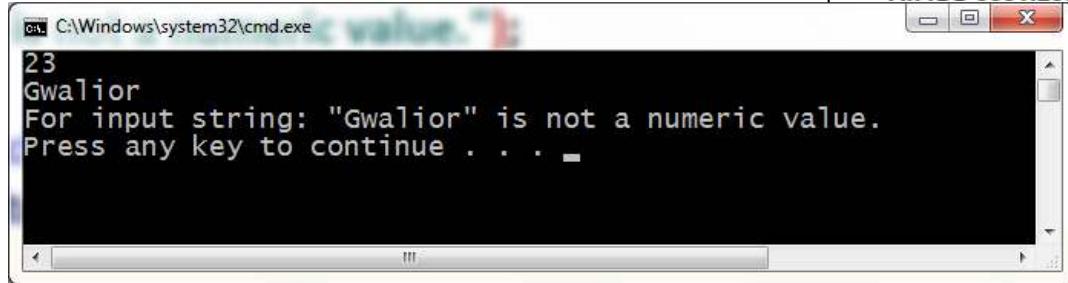
## Syntax

```
try {
// block of code to monitor for errors
}
```

```
catch (ExceptionType1 exOb) {  
// exception handler for ExceptionType1  
  
}  
catch (ExceptionType2 exOb) {  
// exception handler for ExceptionType2  
  
}  
//...  
finally { // block of code to be executed before try block ends  
}
```

In the following example code you will see that how the exception handling can be done in java program. This example reads two integer numbers for the variables a and b. If you enter any other character except number ( 0 - 9 ) then the error is caught by *NumberFormatException* object. After that ex.getMessage() prints the information about the error occurring causes.

```
*****  
* File: exceptionHandle.java  
* Purpose:Simple Exception Handling Program Generate Arithmetic Error  
*****  
import java.io.*;  
public class exceptionHandle{  
    public static void main(String[] args) throws Exception{  
        try{  
            int a,b;  
            DataInputStream in = new DataInputStream(System.in);  
            a = Integer.parseInt(in.readLine());  
            b = Integer.parseInt(in.readLine());  
        }  
        catch(NumberFormatException ex){  
            System.out.println(ex.getMessage()  
            + " is not a numeric value.");  
        }  
        catch(IOException e)  
{System.out.println(e.getMessage());  
    }  
}}
```



A screenshot of a Windows Command Prompt window titled 'cmd.exe'. The window shows the following text:  
23  
Gwalior  
For input string: "Gwalior" is not a numeric value.  
Press any key to continue . . .

### Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 1. Table 2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. Java defines several other types of exceptions that relate to its various class libraries.

**Table 1(UnChecked Exception)**

| Exception                       | Meaning                                                           |
|---------------------------------|-------------------------------------------------------------------|
| ArithmaticException             | Arithmatic error, such as divide-by-zero.                         |
| ArrayIndexOutOfBoundsException  | Array index is out-of-bounds.                                     |
| ArrayStoreException             | Assignment to an array element of an incompatible type.           |
| ClassCastException              | Invalid cast.                                                     |
| EnumConstantNotPresentException | An attempt is made to use an undefined enumeration value.         |
| IllegalArgumentException        | Illegal argument used to invoke a method.                         |
| IllegalMonitorStateException    | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException           | Environment or application is in incorrect state.                 |
| IllegalThreadStateException     | Requested operation not compatible with current thread state.     |
| IndexOutOfBoundsException       | Some type of index is out-of-bounds.                              |
| NegativeArraySizeException      | Array created with a negative size.                               |
| NullPointerException            | Invalid use of a null reference.                                  |
| NumberFormatException           | Invalid conversion of a string to a numeric format.               |
| SecurityException               | Attempt to violate security.                                      |
| StringIndexOutOfBoundsException | Attempt to index outside the bounds of a string.                  |
| TypeNotPresentException         | Type not found.                                                   |
| UnsupportedOperationException   | An unsupported operation was encountered.                         |

**Table 2(Checked Exception)**

| Exception                  | Meaning                                                                            |
|----------------------------|------------------------------------------------------------------------------------|
| ClassNotFoundException     | Class not found.                                                                   |
| CloneNotSupportedException | Attempt to clone an object that does not implement the <b>Cloneable</b> interface. |
| IllegalAccessException     | Access to a class is denied.                                                       |
| InstantiationException     | Attempt to create an object of an abstract class or interface.                     |
| InterruptedException       | One thread has been interrupted by another thread.                                 |
| NoSuchFieldException       | A requested field does not exist.                                                  |
| NoSuchMethodException      | A requested method does not exist.                                                 |

### **Handling Multiple Catch Clauses**

So far we have seen how to use a single catch block, now we will see how to use more than one catch blocks in a single try block. In java when we handle the exceptions then we can have multiple catch blocks for a particular try block to handle many different kind of exceptions that may be generated while running the program i.e. you can use more than one catch clause in a

single try block however every catch block can handle only one type of exception. this mechanism is necessary when the try block has statement that raise different type of exceptions.

The syntax for using this clause is given below:-

```
try{
???
???
}
catch(<exceptionclass_1> <obj1>){
//statements to handle the
exception
}

catch(<exceptionclass_2> <obj2>){
//statements to handle the
exception
}
catch(<exceptionclass_N> <objN>){
//statements to handle the
exception }
```

When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a matching **catch** block that can handle the exception. If no handler is found, then the exception is dealt with by the default exception handler at the top level.

Lets see an example given below which shows the implementation of multiple catch blocks for a single try block.

```
*****
* File: Multi_Catch.java
* Purpose:Simple Exception Handling Program Throws Multiple Execution
*****
```

```
public class Multi_Catch
{
    public static void main (String args[])
    {
        int array[]={20,10,30};
        int num1=15,num2=0;
        int res=0;

        try
        {
            res = num1/num2;
```

```

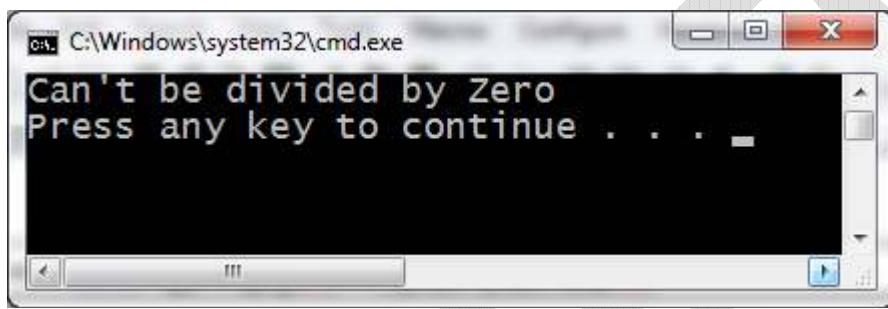
System.out.println("The result is" +res);

for(int ct =2;ct >=0; ct--)
{
    System.out.println("The value of array are" +array[ct]);
}
}

catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error?. Array is out of Bounds");
}

catch (ArithmaticException e)
{
    System.out.println ("Can't be divided by Zero");
}
}

```



In this example we have used two catch clause catching the exception **ArrayIndexOutOfBoundsException** and **ArithmaticException** in which the statements that may raise exception are kept under the try block. When the program is executed, an exception will be raised. Now that time the first catch block is skipped and the second catch block handles the error.

### Nested Try-Catch Blocks

In Java we can have nested **try** and **catch** blocks. It means that, a try statement can be inside the block of another try. If an inner try statement does not have a matching catch statement for a particular exception, the control is transferred to the next try statement's catch handlers that are expected for a matching catch statement. This continues until one of the catch statements succeeds, or until all of the nested try statements are done in. If no one catch statements match, then the Java run-time system will handle the exception.

The syntax of nested **try-catch** blocks is given below:

```

try {
    try {

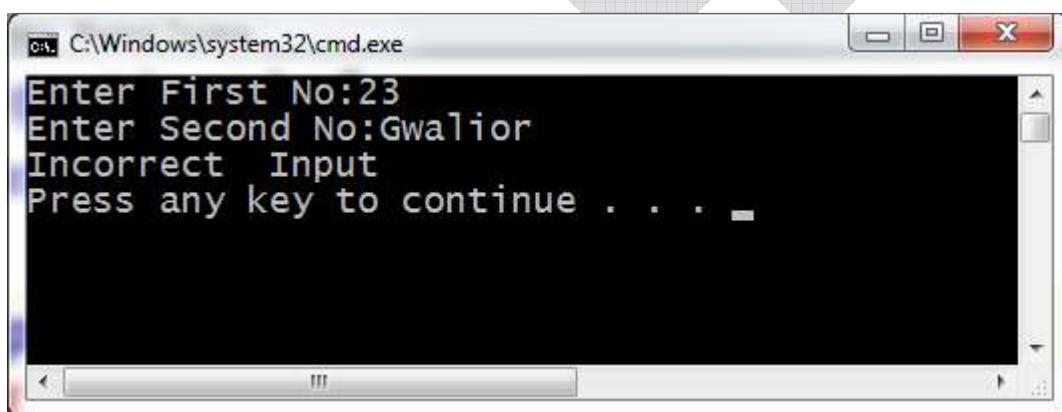
```

```
// ...
}
catch (Exception1 e)
{
//statements to handle the
exception
}
}
catch (Exception 2 e2)
{
//statements to handle the
exception
}
```

```
*****
* File Nested_Try.java
* Purpose:Simple Exception Handling Program Nested try block
*****
import java.io.*;
class Nested_Try
{
    public static void main (String args [ ])
    {
        try
        {DataInputStream X=new DataInputStream(System.in);
         System.out.print("Enter First No:");
         int a = Integer.parseInt (X.readLine());
         System.out.print("Enter Second No:");
         int b = Integer.parseInt (X.readLine());
         int quot = 0;
```

```
try
{
    quot = a / b;
    System.out.println(quot);
}
catch (ArithmetricException e)
{
    System.out.println("divide by zero");
}
catch (NumberFormatException e)
{
    System.out.println ("Incorrect Input");
}
catch (IOException e)
{
    System.out.println ("IO Error");
}
}}
```

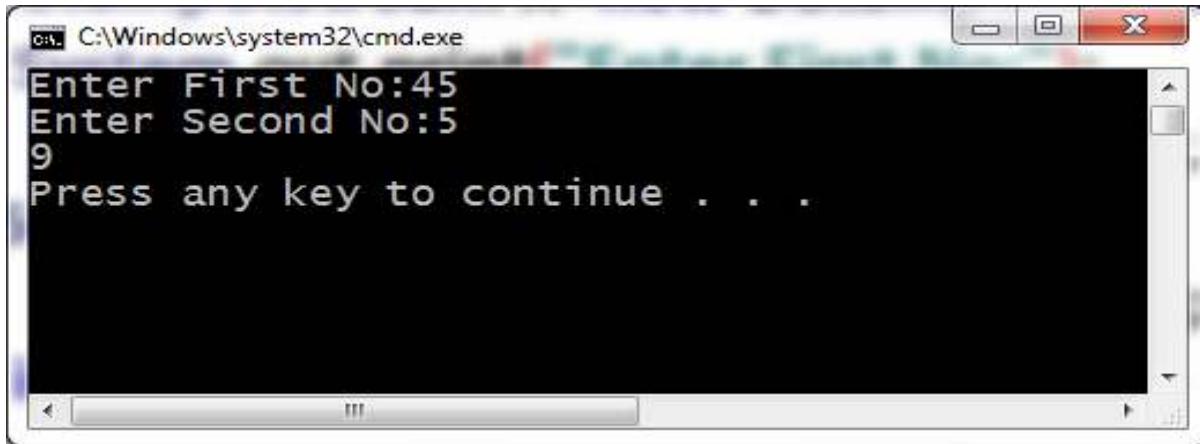
If input is not Integer then an Exception Occur



If input is first no is 23 and next 0 then Error occurs



If input is valid then output is



```
C:\Windows\system32\cmd.exe
Enter First No:45
Enter Second No:5
9
Press any key to continue . . .
```

**DS-SOFECH**

### **throw Keyword**

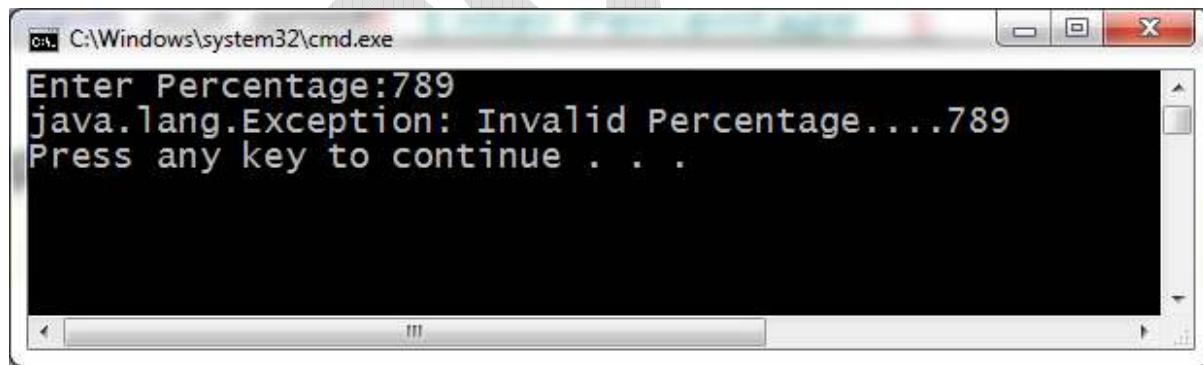
So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

We use throw statement to throw an exception or simply use the throw keyword with an object reference to throw an exception. A single argument is required by the **throw statement i.e. a throwable object.**

```
*****
* File exeexp5.java
* Purpose:Example of throw keyword
*****/
```

```
import java.util.*;
class exexp5
{ public static void main(String arg[])
{ try{
    Scanner KB=new Scanner(System.in);
    System.out.print("Enter Percentage:");
    int per=KB.nextInt();
    if(!(per>=0 && per<=100))
    { throw(new Exception("Invalid Percentage...."+per));
    }
    else
    {System.out.println("Valid Percentage...");}
    }
}catch(Exception e)
{System.out.println(e);
}
}}
```



### throws

"throws" is a keyword defined in the java programming language. Keywords are basically reserved words which have specific meaning relevant to a compiler in java programming language likewise the throw keyword indicates the following :

- ✓ The throws keyword in java programming language is applicable to a method to indicate that the method raises particular type of exception while being processed.
- ✓ The throws keyword in java programming language takes arguments as a list of the objects of type java.lang.Throwable class.

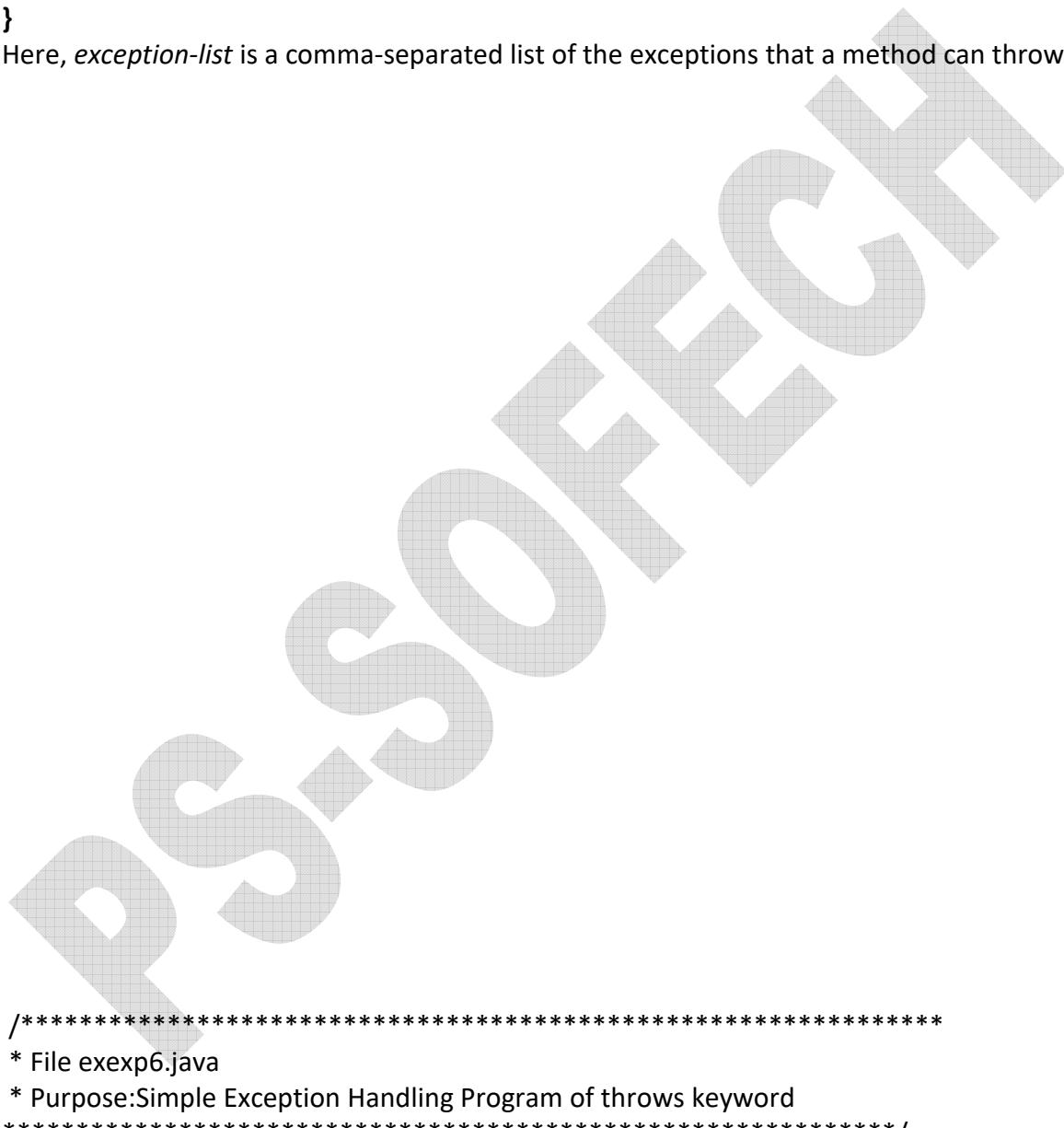
- ✓ When we use the throws with a method it is known as ducking. The method calling a method with a throws clause is needed to be enclosed within the try catch blocks.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
```

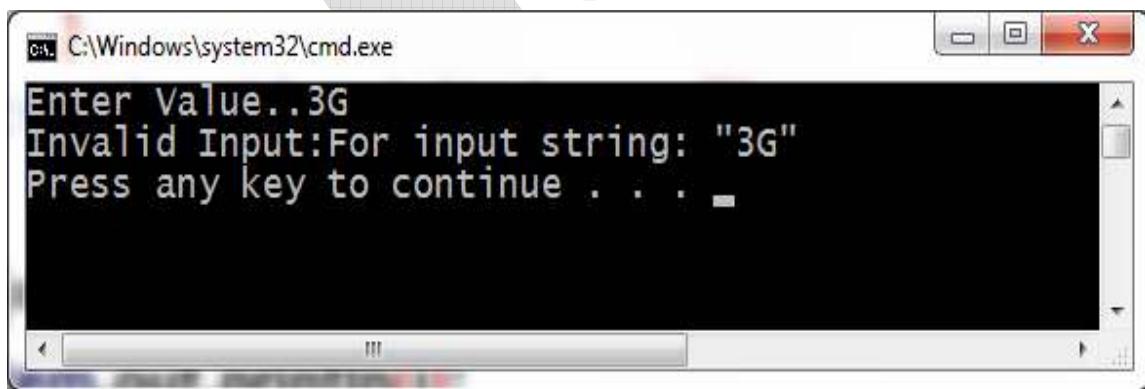
```
{  
// body of method  
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.



```
*****  
* File exexp6.java  
* Purpose:Simple Exception Handling Program of throws keyword  
***** /
```

```
import java.io.*;
class exexp
{ static int division(int x) throws ArithmeticException,NumberFormatException,IOException
    { DataInputStream KB=new DataInputStream(System.in);
        System.out.print("Enter Value..");
        int y=Integer.parseInt(KB.readLine());
        int d=x/y;
        return(d);
    }
    public static void main(String arg[])
    { try{
            int j=division(300);
            System.out.println(j);
        }catch(ArithmetricException e)
        {System.out.println("Error:"+e.getMessage());
        }
        catch(NumberFormatException e)
        {System.out.println("Invalid Input:"+e.getMessage());
        }
        catch(IOException e)
        {System.out.println(e.getMessage());
        }
    }
}
```



## Making Custom (User Defined) Exceptions

So far you would have known, how to be handled the exceptions in java >>>>> 1.3 that are thrown by the Java API but sometimes you may occasionally need to throw your own exception i.e. if you encounter a situation where none of those exception describe your exception accurately or if you can't find the appropriate exception in the Java API, you can code a class that defines an exception that is more appropriate and that mechanism of handling exception is called **Custom** or **User Defined Exception**.

In Java API all exception classes have two type of constructor. First is called default constructor that doesn't accept any arguments. Another constructor accepts a string argument that provides the additional information about the exception. So in that way the Custom exception behaves like the rest of the exception classes in Java API.

***There are two primary use cases for a custom exception.***

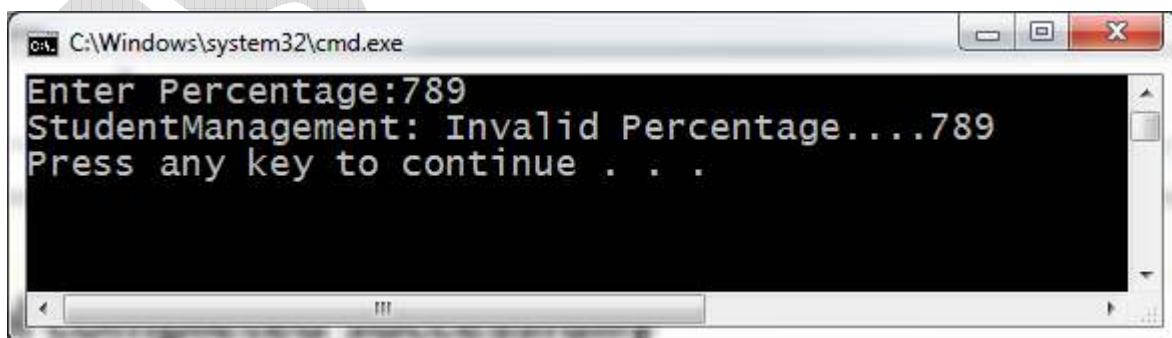
- ***your code can simply throw the custom exception when something goes wrong.***
- ***You can wrap an exception that provides extra information by adding your own message.***

**The code of a Custom exception:**

```
public class ExceptionClassName  
extends Exception  
{  
    public ExceptionClassName(){ }  
    public  
ExceptionClassName(StringMessage)  
    {  
        super(message);  
    }  
}
```

```
*****
* File exexp7.java
* Purpose: Making our own Exception
*****/
```

```
import java.util.*;
class StudentManagement extends Exception
{ StudentManagement(String errmsg)
  {super(errmsg);
  }
}
class exexp7
{public static void main(String arg[])
 { try{
    Scanner KB=new Scanner(System.in);
    System.out.print("Enter Percentage:");
    int per=KB.nextInt();
    if(!(per>=0 && per<=100))
     { throw(new StudentManagement("Invalid Percentage...."+per));
      }
    else
     {System.out.println("Valid Percentage... ");
      }
   }catch(StudentManagement e)
 {System.out.println(e);
  }
 }}
```



## **Finally Block**

*It is always a good practice to use finally clause after the try and catch block to handle an unexpected exception occurred in the try block. It becomes possible because the finally block always executes when the try block exits and it is also useful to write the cleanup code within the finally block. The finally block is executed surely whether the exception is generated or not. This is a Java keyword therefore it may not be used as identifiers i.e. you cannot declare a variable or class with this name in your Java program. It is used when the finally block is executed after the execution exits the try block and any associated catch clauses regardless of whether an exception was thrown or caught.*

The syntax of declaring a final type variable is:

```
try {  
    / Normal execution path  
  
} catch (ExampleException ee) {  
    / deal with the ExampleException  
}  
  
finally {  
    / This optional section is executed upon  
    / termination of any of the try or catch blocks  
    /above  
}
```

[PS-SOFTECH,Gwalior]

# [Java -(Part Two)]

[Threads,IO Streams,JDBC,Socket,RMI,Applets &  
Swings,Events]

Compiled By Sandeep Sappal

# Index

|                                                      |    |
|------------------------------------------------------|----|
| Exception Propagation.....                           | 2  |
| Life Cycle Of Thread.....                            | 5  |
| Thread Creation .....                                | 8  |
| Thread Constructor.....                              | 8  |
| Multithreading.....                                  | 9  |
| Creation Of Multiple Thread.....                     | 10 |
| Deadlock.....                                        | 11 |
| Inter Thread Communication(Producer & Consumer)..... | 17 |
| I/O Stream.....                                      | 24 |
| What is Java I/O.....                                | 25 |
| Using a Random Access File.....                      | 32 |
| Character Streams.....                               | 36 |
| Java Writer.....                                     | 38 |
| Serializing an object.....                           | 40 |
| De Serializing an object.....                        | 42 |
| Other Useful I/O Stream class.....                   | 43 |
| Working with File.....                               | 46 |
| JAVA Directory.....                                  | 47 |
| JAVA Database Connectivity.....                      | 51 |
| Connection Interface.....                            | 55 |
| Statement Interface.....                             | 55 |
| Result Set Interface.....                            | 56 |
| ResultSet Metadata Interface.....                    | 60 |
| PreparedStatement Interface.....                     | 60 |
| CallableStatement Interface.....                     | 62 |

|                                    |     |
|------------------------------------|-----|
| Explanation Of JDBC Steps.....     | 62  |
| Exception on SQL.....              | 63  |
| Applet(Life Cycle of Applets)..... | 71  |
| AWT Containers Component.....      | 74  |
| What is AWT in JAVA.....           | 74  |
| Containers Vs Component.....       | 77  |
| AWT Hierarchy of Class.....        | 77  |
| AWT Control with Examples.....     | 79  |
| Applet class.....                  | 80  |
| Graphics class.....                | 81  |
| Drawing images.....                | 82  |
| Drawing ovals.....                 | 83  |
| Drawing rounded rectangles.....    | 86  |
| Drawing 3D rectangles.....         | 87  |
| Drawing strings.....               | 88  |
| Java color example.....            | 88  |
| Setting the drawing color.....     | 89  |
| Setting text fonts.....            | 91  |
| Java AWT checkbox examples.....    | 97  |
| Java AWT choice examples.....      | 101 |
| Java AWT Label examples.....       | 103 |
| Java AWT List examples.....        | 104 |
| Java AWT Scrollbar examples.....   | 106 |
| Java AWT Text Area example.....    | 109 |
| BorderLayout.....                  | 112 |
| FowLayout.....                     | 114 |

|                                                       |     |
|-------------------------------------------------------|-----|
| GridLayout.....                                       | 115 |
| Menus in applets.....                                 | 120 |
| Swings.....                                           | 125 |
| Jframe.....                                           | 128 |
| Jpanel.....                                           | 130 |
| Jbutton.....                                          | 133 |
| Jcheckbox.....                                        | 135 |
| Jlabel.....                                           | 143 |
| JpasswordField.....                                   | 144 |
| JradioButton.....                                     | 146 |
| Jslider.....                                          | 151 |
| JtextArea.....                                        | 154 |
| JtextField.....                                       | 155 |
| BoxLayout.....                                        | 156 |
| GridLayout.....                                       | 158 |
| Networking socket.....                                | 162 |
| Socket.....                                           | 168 |
| UDP.....                                              | 174 |
| Datagram .....                                        | 174 |
| R M I(Remote Method Invocation).....                  | 177 |
| Events.....                                           | 181 |
| AWT events.....                                       | 182 |
| Delegation of event handling to listener objects..... | 183 |



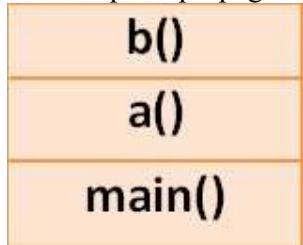
### **Exception propagation : Propagation of Uncaught exceptions**

There are many important things to remember to catch an exception. For instance one must use catch or finally to catch the thrown exceptions in try block. Also remember that try will

never work without either a catch or finally. This is a must follow condition.

So what happens to a exception which is not caught by any of the catch block, of course they keep looking for its handler (its always good to catch 'em) and if they failed to find one they just propagate from one method to the other and end up crashing the program. Whenever an exception is not caught by any catch block in the method then method is said to be ducking the exception. It also means that method passes its course of work to other method to handle the exception. The exception propagation follows call stack. I think you are now grown up and must know what a call stack means. Lets rewind the stack concept that a call stack is a stack where methods are placed in LIFO order based on method call. For example if yo have called method abc() from main() then main() method will be placed at bottom of the stack and the called method abc() will be on top. After the abc finish execution the abc() will be popped from the stack and main() will take pole position.

The above call stack concept also applies to the Exception propagation. Consider we have two methods a() and b(). main() calls a() and after that a() calls method b(). Now suppose the called method b() is facing heat from JVM and it resulted in exception then the exception will look for specific handler, if it failed to either specific handler or a superclass handler then it start to look in the calling method that is a(). Here also it will follow the same procedure and if it again failed it will go to main() method and in the end if it again failed at the bottom of the stack then mate its ultimate law of nature that it has to die a horrible death, or it will produce a complete stack trace of the exception. Following figure shows how call stack works and exception propagate:-



Make it even more clear by considering the following example:-

```
public class Exceptionpropagation{
public static void main(String[] a){
domore();
}
static public void domore(){
doevenmore();
}
static public void doevenmore(){
int x=10/0;
}
}
```

The produced will be in the order of stack call that is from code it is clear that the exception is generated from doevenmore() so it will be listed on top of stack trace and following the stack trace you can find out where actually the exception has happened.

The resulting exception would be and its stack trace :-

*Exception in thread "main"*

```
java.lang.ArithmetricException: / by zero  
at Exceptionpropagation.doevenmore(Exceptionpropagation.java:13)  
at Exceptionpropagation.domore(Exceptionpropagation.java:10)  
at Exceptionpropagation.main(Exceptionpropagation.java:7)  
Process Exit...
```



# MultiThreads



## Process

A **process** is an instance of a computer program that is executed sequentially. It is a collection of instructions which are executed simultaneously at the run time. Thus several processes may be associated with the same program. For example, to check the **spelling** is a single process in the **Word Processor** program and you can also use other processes like **printing, formatting, drawing, etc.** associated with this program.

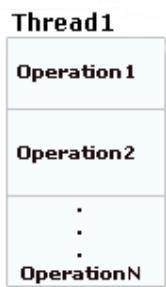
## Thread

A thread is a **lightweight** process which exists within a program and executes to perform a special task. Several threads of execution may be associated with a single process. Thus a

process that has only one thread is referred to as a **single-threaded** process, while a process with multiple threads is referred to as a **multi-threaded** process.

In Java Programming language, thread is a sequential path of code execution within a program. Each thread has its own local variables, program counter and lifetime. In single threaded runtime environment, operations are executed sequentially i.e. next operation can execute only when the previous one is complete. It exists in a common memory space and can share both data and code of a program. Threading concept is very important in Java through which we can increase the speed of any application. You can see diagram shown below in which a thread is executed along with its several operations with in a single process.

#### Single Process



#### Main Thread

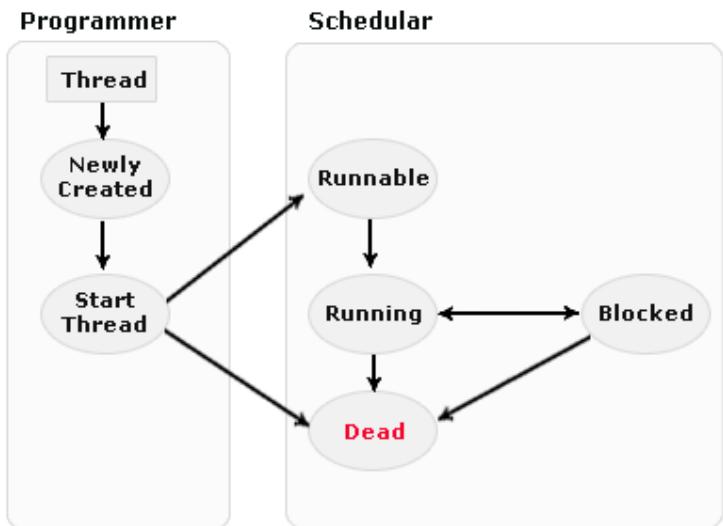
When any standalone application is running, it firstly executes the **main()** method in a single thread, called the main thread. If no other threads are created by the main thread, then the program terminates when the **main()** method completes its execution. The main thread creates some other threads called child threads. The **main()** method execution can finish, but the program will keep running until all threads have completed their execution.

**When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states. By invoking start() method, it doesn't mean that the thread has access to CPU and starts executing straight away. Several factors determine how it will proceed.**

#### Life Cycle of A Thread

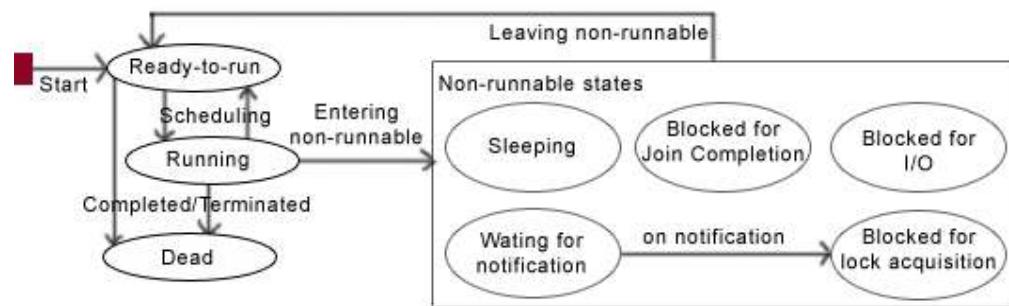
When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states. By invoking **start()** method, it doesn't mean that the thread has access to CPU and starts executing straight away. Several factors determine how it will proceed.

#### Different states of a thread are :



1. **New state** – After the creation of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
2. **Runnable (Ready-to-run) state** – A thread starts its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
3. **Running state** – A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler selects a thread from runnable pool.
4. **Dead state** – A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
5. **Blocked** - A thread can enter in this state because of waiting for resources that are held by another thread.

#### Different states implementing Multiple-Threads are:



As we have seen different states that may occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enter directly to the running state from non-runnable state, firstly it goes to runnable state. Now let's understand the some non-runnable states which may occur handling the multithreads.

- **Sleeping** – On this state, the thread is still alive but it is not runnable, it might be returned to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method sleep() to stop the running state of a thread.

**static void sleep(long millisecond) throws InterruptedException**

- **Waiting for Notification** – A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.

**final void wait(long timeout) throws InterruptedException**

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

**final void wait() throws InterruptedException**

- **Blocked on I/O** – The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.
- **Blocked for joint completion** – The thread can come on this state because of waiting the completion of another thread.
- **Blocked for lock acquisition** – The thread can come on this state because of waiting to acquire the lock of an object.

#### Methods that can be applied apply on a Thread:

Some Important Methods defined in **java.lang.Thread** are shown in the table:

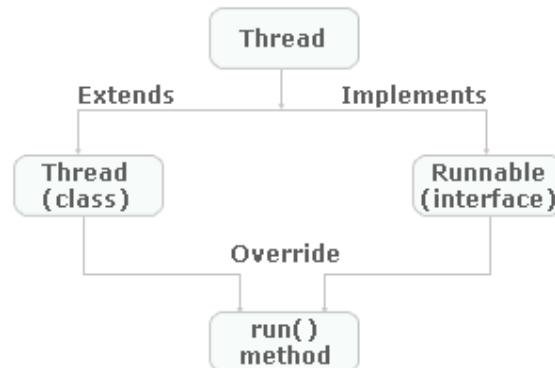
| Method          | Return Type | Description                                                                                                                                                                                                      |
|-----------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| currentThread() | Thread      | Returns an object reference to the thread in which it is invoked.                                                                                                                                                |
| getName()       | String      | Retrieve the name of the thread object or instance.                                                                                                                                                              |
| start()         | void        | Start the thread by calling its run method.                                                                                                                                                                      |
| run()           | void        | This method is the entry point to execute thread, like the main method for applications.                                                                                                                         |
| sleep(int n )   | void        | Suspends a thread for a specified amount of time (in milliseconds).                                                                                                                                              |
| isAlive()       | boolean     | This method is used to determine the thread is running or not.                                                                                                                                                   |
| activeCount()   | int         | This method returns the number of active threads in a particular thread group and all its subgroups.                                                                                                             |
| interrupt()     | void        | The method interrupt the threads on which it is invoked.                                                                                                                                                         |
| yield()         | void        | By invoking this method the current thread pause its execution temporarily and allow other threads to execute.                                                                                                   |
| join()          | void        | This method and <b>join(long millisec)</b> Throws InterruptedException. These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively. |
| suspend()       | void        | Pause the thread process                                                                                                                                                                                         |
| resume()        | void        | Restart the suspended process                                                                                                                                                                                    |
| getName()       | String      | Returns the thread name                                                                                                                                                                                          |
| setName(String) | void        | Assign Thread name                                                                                                                                                                                               |
| getPriority()   | int         | Returns Thread priority                                                                                                                                                                                          |

|             |      |                                              |
|-------------|------|----------------------------------------------|
| wait()      | int  | Pause the thread in synchronization(locking) |
| notify()    | void | Unlock wait process                          |
| notifyAll() | void | Unloack all Thread Process                   |

## Thread Creation

In Java, an object of the Thread class can represent a thread. Thread can be implemented through any one of two ways:

- Extending the **java.lang.Thread Class**
- Implementing the **java.lang.Runnable Interface**



### I. Extending the **java.lang.Thread Class**

For creating a thread a class have to extend the Thread Class. For creating a thread by this procedure you have to follow these steps:

1. Extend the **java.lang.Thread Class**.
2. Override the **run( )** method in the subclass from the Thread class to define the code executed by the thread.
3. Create an **instance** of this subclass. This subclass may call a Thread class constructor by subclass constructor.
4. Invoke the **start( )** method on the instance of the class to make the thread eligible for running.

### II. Implementing the **java.lang.Runnable Interface**

The procedure for creating threads by implementing the Runnable Interface is as follows:

1. A Class implements the **Runnable Interface**, override the **run()** method to define the code executed by thread. An object of this class is Runnable Object.
2. Create an object of **Thread Class** by passing a Runnable object as argument.
3. Invoke the **start( )** method on the instance of the Thread class.

## Thread Constructors

Several constructors are available for creating new Thread instances.

**Thread(): Default Constructor**

**Thread(String thread name) :** This constructor create Thread object with specified thread name.

**Thread(Runnable) :** This constructor used when user create thread object with Runnable interface to control thread

# Multithreading in Java

## Introduction

So far you have learned about a single thread. Lets us know about the concept of multithreading and learn the implementation of it. But before that, lets be aware from the multitasking.

### Multitasking :

Multitasking allow to execute more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time. Multitasking allow processes (i.e. programs) to run concurrently on the program. For Example running the [spreadsheet](#) program and you are working with word processor also.

Multitasking is running heavyweight processes by a single OS.

### Multithreading :

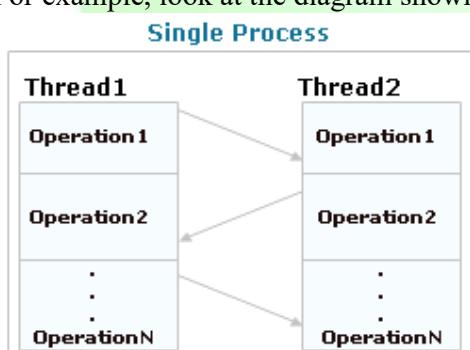
Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system

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.

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.

### 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.

# Creation of Multiple Threads

## Thread Priorities

In Java, thread scheduler can use the thread **priorities** in the form of **integer value** to each of its thread to determine the execution schedule of threads . Thread gets the **ready-to-run** state according to their priorities. The **thread scheduler** provides the CPU time to thread of highest priority during ready-to-run state.

Priorities are integer values from 1 (lowest priority given by the constant **Thread.MIN\_PRIORITY**) to 10 (highest priority given by the constant **Thread.MAX\_PRIORITY**). The default priority is 5(**Thread.NORM\_PRIORITY**).

| Constant             | Description                                             |
|----------------------|---------------------------------------------------------|
| Thread.MIN_PRIORITY  | The maximum priority of any thread (an int value of 10) |
| Thread.MAX_PRIORITY  | The minimum priority of any thread (an int value of 1)  |
| Thread.NORM_PRIORITY | The normal priority of any thread (an int value of 5)   |

The methods that are used to set the priority of thread shown as:

| Method        | Description                                           |
|---------------|-------------------------------------------------------|
| setPriority() | This is method is used to set the priority of thread. |
| getPriority() | This method is used to get the priority of thread.    |

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

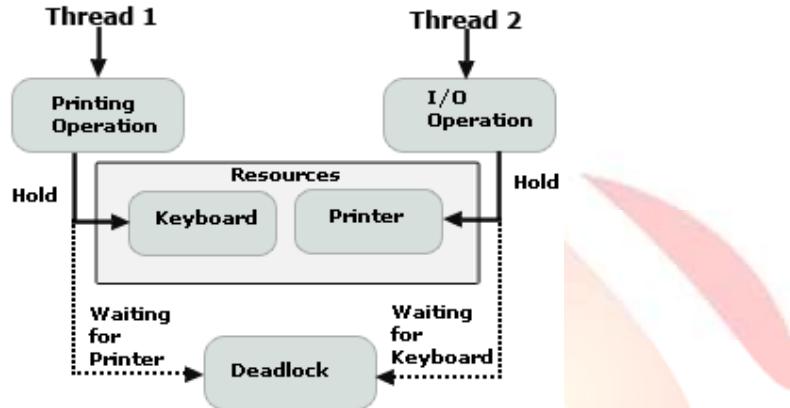
## Thread Scheduler

In the implementation of threading scheduler usually applies one of the two following strategies:

- **Preemptive scheduling** – If the new thread has a higher priority than current running thread leaves the runnable state and higher priority thread enter to the runnable state.
- **Time-Sliced (Round-Robin) Scheduling** – A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.

## Deadlock

- A situation where a thread is waiting for an object lock that holds by second thread, and this second thread is waiting for an object lock that holds by first thread, this situation is known as **Deadlock**.
- Lets see a situation in the diagram shown below where the deadlock condition is occurred :



- In this diagram two threads having the **Printing & I/O** operations respectively at a time. But **Thread1** need to **printer** that is hold up by the **Thread2**, likewise **Thread2** need the **keyboard** that is hold up by the **Thread1**. In this situation the CPU becomes idle and the deadlock condition occurs because no one thread is executed until the hold up resources are free.
- **The following program demonstrates the deadlock situation:**

```
public class DeadDemo{
    public static void main(String args[]){
        String s1="Dead";
        String s2="Lock";
        MyThread1 m=new MyThread1(s1,s2);
        MyThread2 m1=new MyThread2(s1,s2);
    }
}

class MyThread1 extends Thread{
    String s1;
    String s2;
    MyThread1(String s1, String s2){
        this.s1=s1;
        this.s2=s2;
        start();
    }

    public void run(){
        while(true){
            synchronized(s1){

                synchronized(s2){
                    System.out.println(s1+s2);
                }
            }
        }
    }
}

class MyThread2 extends Thread{
    String s1;
```

```
String s2;
MyThread2(String s1, String s2) {
    this.s1=s1;
    this.s2=s2;
    start();
}

public void run() {
    while(true) {
        synchronized(s2) {
            synchronized(s1) {
                System.out.println(s2+s1);
            }
        }
    }
}
```

**Output of the program is:**

```
C:\j2se6\thread>javac DeadDemo.java
```

```
C:\j2se6\thread>java DeadDemo
```

```
DeadLock
```

```
LockDead
```

```
DeadLock
```

```
.....
```

```
.....
```

Java, the threads are executed independently to each other. These types of threads are called as **asynchronous threads**. But there are two problems may be occur with asynchronous threads.

- Two or more threads share the same resource (variable or method) while only one of them can access the resource at one time.
- If the producer and the consumer are sharing the same kind of data in a program then either producer may produce the data faster or consumer may retrieve an order of data and process it without its existing.

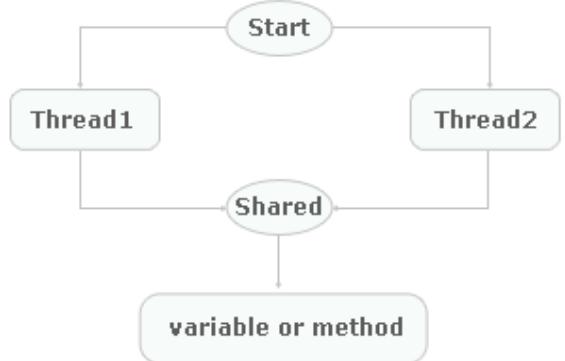
Suppose, we have created two methods as **increment()** and **decrement()**, which increases or decreases value of the variable "**count**" by 1 respectively shown as:

```
public void
increment( ) {
    count++;
}

public void
decrement( ) {
    count--;
}
public int value()
{
    return
count;
}
```

When the two threads are executed to access these methods (one for **increment()**, another for **decrement()**) then both will share the variable "**count**". In that case, we can't be sure that what value will be returned of variable "**count**".

We can see this problem in the diagram shown below:



To avoid this problem, Java uses **monitor** also known as "**semaphore**" to prevent data from being corrupted by multiple threads by a keyword **synchronized** to synchronize them and intercommunicate to each other. It is basically a mechanism which allows two or more threads to share all the available resources in a sequential manner. Java's synchronized is used to ensure that only one thread is in a **critical region**. Critical region is a lock area where only one thread is run (or lock) at a time. Once the thread is in its critical section, no other thread can enter to that critical region. In that case, another thread will have to wait until the current thread leaves its critical section.

General form of the synchronized statement is as:

```
synchronized(object)
{
    // statements to be
    synchronized
}
```

### Lock:

**Lock** 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 synchronized.
- 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

**There are two ways to synchronize the execution of code:**

**1. Synchronized Methods**

**2. Synchronized Blocks (Statements)**

**Synchronized Methods:**

Any method is specified with the keyword **synchronized** is only executed by one thread at a time. If any thread want to execute the synchronized method, firstly it has to obtain the objects lock. If the lock is already held by another thread, then calling thread has to wait.

Synchronized methods are useful in those situations where methods are executed concurrently, so that these can be intercommunicate manipulate the state of an object in ways that can corrupt the state if . Stack implementations usually define the two operations push and pop of elements as synchronized, that's why pushing and popping are mutually exclusive operations. For Example if several threads were sharing a stack, if one thread is popping the element on the stack then another thread would not be able to pushing the element on the stack.

**The following program demonstrates the synchronized method:**

```
class Share extends Thread{
    static String msg[]{"This", "is", "a", "synchronized", "variable"};
    Share(String threadname){
        super(threadname);
    }
    public void run(){
        display(getName());
    }
    public synchronized void display(String threadN){
        for(int i=0;i<=4;i++)
            System.out.println(threadN+msg[i]);
        try{
            this.sleep(1000);
        }catch(Exception e){}
    }
}
public class SynThread1 {
    public static void main(String[] args) {
        Share t1=new Share("Thread One: ");
        t1.start();
        Share t2=new Share("Thread Two: ");
        t2.start();
    }
}
```

**Output of the program is:**

```
C:\nisha>javac
SynThread.java
```

```
C:\nisha>java
SynThread
Thread One: This
Thread One: is
Thread One: a
Thread One:
synchronized
```

```
Thread One: variable  
Thread Two: This  
Thread Two: is  
Thread two: a  
Thread Two:  
synchronized  
Thread Two: variable
```

In this program, the method "**display( )**" is synchronized that will be shared by both thread's objects at the time of program execution. Thus only one thread can access that method and process it until all statements of the method are executed.

### Synchronized Blocks (Statements)

Another way of handling synchronization is Synchronized Blocks (Statements).

Synchronized statements must specify the object that provides the native lock. The synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.

General form of synchronized block is:

```
synchronized (object reference  
expression)  
{  
// statements to be  
synchronized  
}
```

The following program demonstrates the **synchronized block** that shows the same output as the output of the previous example:

```
class Share extends Thread{  
    static String msg[]={ "This", "is", "a", "synchronized", "variable" };  
    Share(String threadname){  
        super(threadname);  
    }  
    public void run(){  
        display(getName());  
    }  
    public void display(String threadN){  
        synchronized(this){  
            for(int i=0;i<=4;i++)  
                System.out.println(threadN+msg[i]);  
            try{  
                this.sleep(1000);  
            }catch(Exception e){}  
        }  
    }  
}  
public class SynStatement {  
    public static void main(String[] args) {  
        Share t1=new Share("Thread One: ");  
        t1.start();  
        Share t2=new Share("Thread Two: ");  
        t2.start();  
    }  
}
```

### Output of the Program

```
C:\nisha>javac  
SynStatement.java
```

```
C:\nisha>java  
SynStatement  
Thread One: This  
Thread One: is  
Thread One: a  
Thread One:  
synchronized  
Thread One: variable  
Thread Two: This  
Thread Two: is  
Thread Two: a  
Thread Two:  
synchronized  
Thread Two: variable
```

## Inter-Thread Communication

Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU's idle time i.e. A process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed. This technique is known as **Interthread communication** which is implemented by some methods. These methods are defined in "java.lang" package and can only be called within synchronized code shown as:

| Method       | Description                                                                                                                                                                     |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| wait( )      | It indicates the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls method <b>notify()</b> or <b>notifyAll()</b> . |
| notify( )    | It wakes up the first thread that called <b>wait()</b> on the same object.                                                                                                      |
| notifyAll( ) | Wakes up (Unlock) all the threads that called <b>wait()</b> on the same object. The highest priority thread will run first.                                                     |

All these methods must be called within a try-catch block.

Lets see an example implementing these methods :

```
class Shared {  
  
    int num=0;  
    boolean value = false;  
  
    synchronized int get() {  
        if (value==false)  
            try {  
                wait();  
            }  
        catch (InterruptedException e) {  
            System.out.println("InterruptedException caught");  
        }  
        System.out.println("consume: " + num);  
        value=false;  
        notify();  
        return num;  
    }  
}
```

```

synchronized void put(int num) {
    if (value==true)
        try {
            wait();
        }
    catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
    }
    this.num=num;
    System.out.println("Produce: " + num);
    value=false;
    notify();
}
}

class Producer extends Thread {
Shared s;

Producer(Shared s) {
    this.s=s;
    this.start();
}

public void run() {
    int i=0;

    s.put(++i);
}
}

class Consumer extends Thread{
Shared s;

Consumer(Shared s) {
    this.s=s;
    this.start();
}

public void run() {
    s.get();
}
}

public class InterThread{
    public static void main(String[] args)
    {
        Shared s=new Shared();
        new Producer(s);
        new Consumer(s);
    }
}

```

#### **Output of the Program:**

```
C:\nisha>javac
InterThread.java
```

```
C:\nisha>java
InterThread
Produce: 1
consume: 1
```

In this program, two threads "**Producer**" and "**Consumer**" share the `synchronized` methods of the class "**Shared**". At time of program execution, the "`put( )`" method is invoked through the "**Producer**" class which increments the variable "`num`" by 1. After producing 1 by the producer, the method "`get( )`" is invoked by through the "**Consumer**" class which retrieves the produced number and returns it to the output. Thus the Consumer can't retrieve the number without producing of it.

### [Download this Program](#)

**Another program demonstrates the uses of `wait()` & `notify()` methods:**

```
public class DemoWait extends Thread{
    int val=20;
    public static void main(String args[]) {
        DemoWait d=new DemoWait();
        d.start();
        new Demo1(d);
    }
    public void run(){
        try {
            synchronized(this) {
                wait();
                System.out.println("value is :"+val);
            }
        }catch(Exception e){}
    }

    public void valchange(int val){
        this.val=val;
        try {
            synchronized(this) {
                notifyAll();
            }
        }catch(Exception e){}
    }
}
class Demo1 extends Thread{
    DemoWait d;
    Demo1(DemoWait d) {
        this.d=d;
        start();
    }
    public void run(){
        try{
            System.out.println("Demo1 value is"+d.val);
            d.valchange(40);
        }catch(Exception e){}
    }
}
```

**Output of the program is:**

```
C:\j2se6\thread>javac
DemoWait.java
```

```
C:\j2se6\thread>java  
DemoWait  
Demo1 value is20  
value is :40
```

```
C:\j2se6\thread>
```

### Daemon thread

In Java, any thread can be a Daemon thread. Daemon threads are like a **service providers** for other threads or objects running in the same process as the daemon thread. Daemon threads are used for background supporting tasks and are only needed while normal threads are executing. If normal threads are not running and remaining threads are daemon threads then the interpreter exits.

**setDaemon(true/false)** – This method is used to specify that a thread is daemon thread.

**public boolean isDaemon()** – This method is used to determine the thread is daemon thread or not.

The following program demonstrates the **Daemon Thread**:

```
public class DaemonThread extends Thread {  
    public void run() {  
        System.out.println("Entering run method");  
  
        try {  
            System.out.println("In run Method: currentThread() is"  
                + Thread.currentThread());  
  
            while (true) {  
                try {  
                    Thread.sleep(500);  
                } catch (InterruptedException x) {  
                }  
  
                System.out.println("In run method: woke up again");  
            }  
        } finally {  
            System.out.println("Leaving run Method");  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println("Entering main Method");  
  
        DaemonThread t = new DaemonThread();  
        t.setDaemon(true);  
        t.start();  
  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException x) {  
        }  
  
        System.out.println("Leaving main method");  
    }  
}
```

```
}
```

**Output of this program is:**

```
C:\j2se6\thread>javac  
DaemonThread.java
```

```
C:\j2se6\thread>java DaemonThread  
Entering main Method  
Entering run method  
In run Method: currentThread()  
isThread[Thread-0,main]  
In run method: woke up again  
Leaving main method
```

```
C:\j2se6\thread>
```

### **Example of sleep(),stop() and yield() method.**

**Program :ch.java**

```
class india extends Thread  
{ india(String name)//constructor to assign thread name  
    { super(name); //call base class constructor  
    }  
    public void run() //thread process  
    { for(int i=1;i<=5;i++)  
        { if(i==3)  
            { System.out.println("India Relinquish Control Here... ");  
                yield();}  
            System.out.println(getName());  
        } }  
class china extends Thread  
{ china(String name)  
    { super(name); }  
    public void run()  
    { for(int i=1;i<=5;i++)  
        { try{sleep(500); //sleep for 500 ms.  
        }  
        catch(InterruptedException e)  
        {System.out.println(e); }  
        System.out.println(getName());  
    } }  
  
class usa extends Thread  
{ usa(String name)
```

```

    { super(name); }
    public void run()
    {for(int i=1;i<=5;i++)
    {
        if(i==2)
        { stop();//stop the thread
        }
        System.out.println(getName());
    }}
class th
{ public static void main(String arg[])
{ india i=new india("MyIndia");
china c=new china("Yours China");
usa u=new usa("Obama's USA");
i.start();
c.start();
u.start();}
}
output
MyIndia
MyIndia
India Relinquish Control Here...
Obama's USA
MyIndia
MyIndia
MyIndia
Yours China
Yours China
Yours China
Yours China
Yours China
Press any key to continue . .

```

### **Example of suspend(),resume() method**

#### **Program spellchk.java**

```

import java.io.*;
class spell extends Thread
{ String word=new String();
    public void run() //spell checker check countries
    { while(true)
        { if(word.equals("india"))
            { System.out.println("India Tested Ok");
            }
            else if(word.equals("usa"))
            { System.out.println("USA Tested Ok");
            }
            else if(word.equals("exit"))
            { stop();
            suspend();// pause the process
            }
        }
    }
class spellchk
{     public static void main(String arg[])
{   spell w=new spell();
    w.start();
    do{
        try{
            DataInputStream X=new DataInputStream(System.in);
            System.out.print("Enter Word:");
            w.word=X.readLine();
            w.resume();//restart the process
        }catch(Exception e)
        {System.out.println(e);}
        }while(!w.word.equals("exit"));
    }
}

```

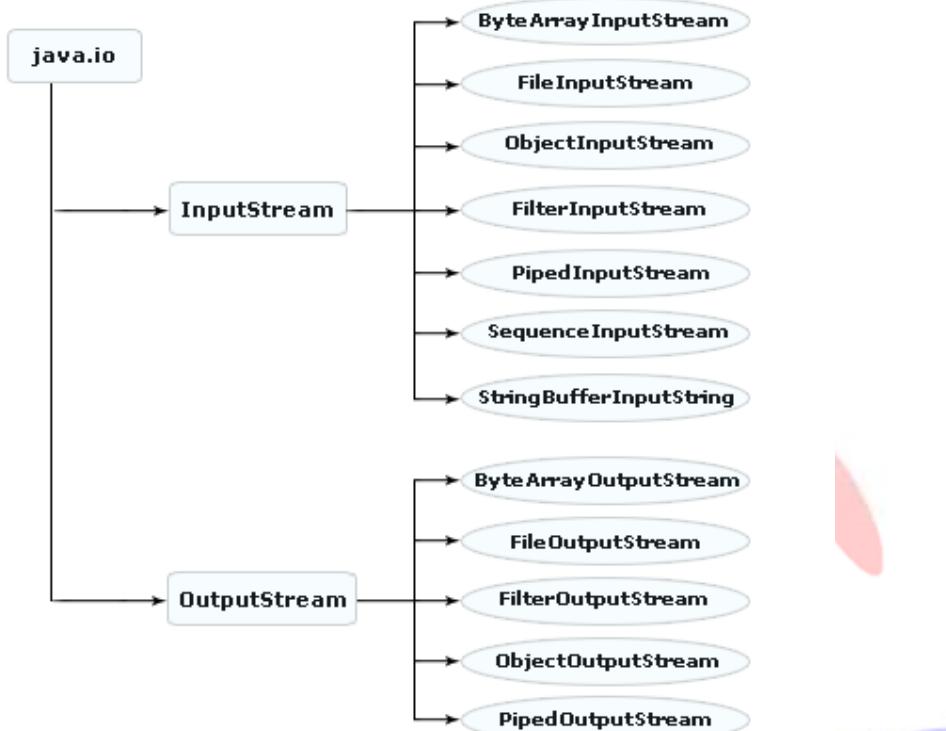


## What is Java I/O?

### *Introduction*

The Java Input/Output (I/O) is a part of **java.io** package. The **java.io** package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources. The **InputStream** and **OutputStream** are central classes in the package which are used for reading from and writing to byte streams, respectively.

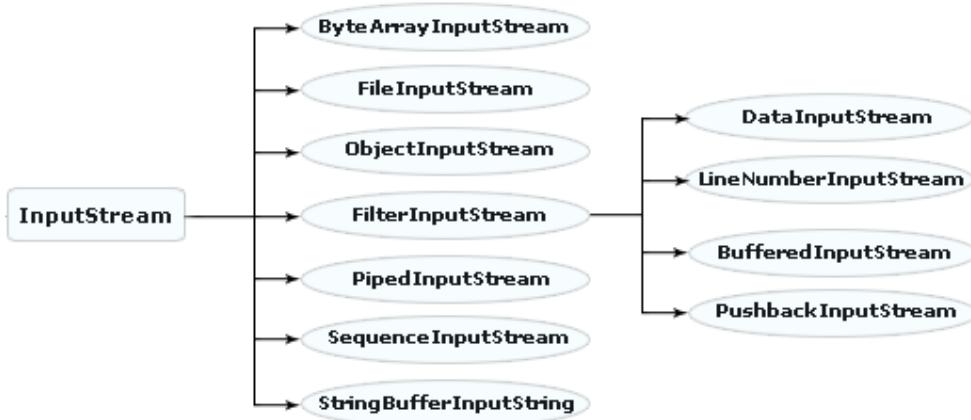
The **java.io** package can be categorized along with its stream classes in a hierarchy structure shown below:



### **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.

The subclasses inherited from the **InputStream** class can be seen in a hierarchy manner shown below:

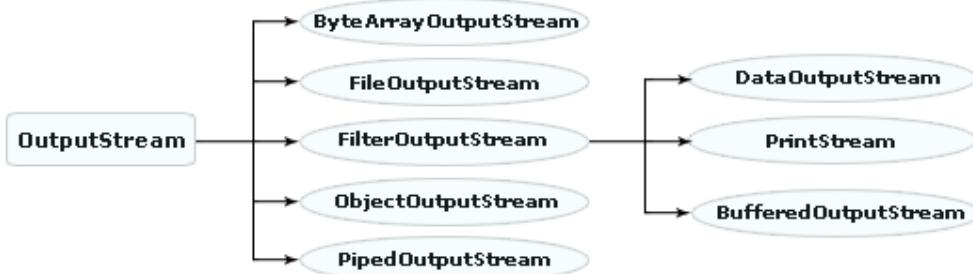


InputStream is inherited from the Object class. Each class of the InputStreams provided by the java.io package is intended for a different purpose.

### **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. The classes inherited from the **OutputStream** class can be seen in a hierarchy structure shown below:

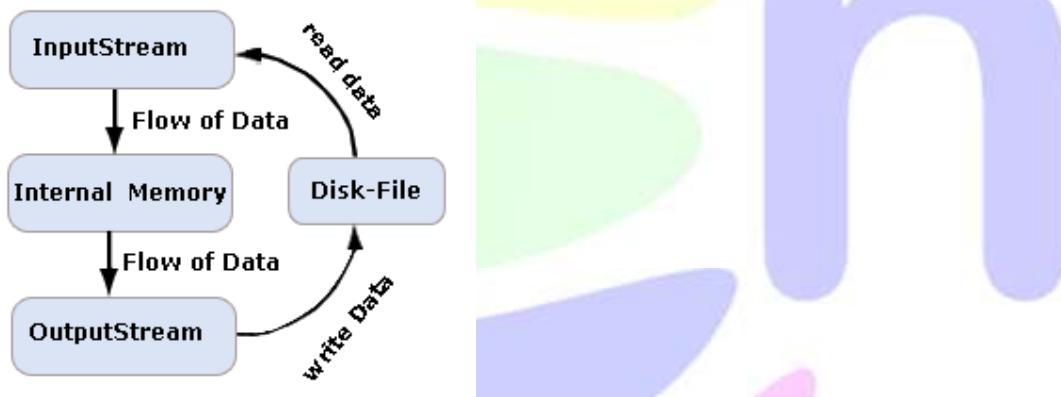


OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the java.io package is intended for a different purpose.

### 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.

The working process of the I/O streams can be shown in the given diagram.



### Some Common Methods of InputStream:

1) `public abstract int read() throws IOException`

Reads a byte of data. This method will block if no input is available.

#### Returns:

the byte read, or -1 if the end of the stream is reached.

#### Throws: IOException

If an I/O error has occurred.

2) `public int read(byte b[],  
int off,  
int len) throws IOException`

Reads into an array of bytes. This method will block until some input is available.

**Parameters:**

b - the buffer into which the data is read  
off - the start offset of the data  
len - the maximum number of bytes read

**Returns:**

the actual number of bytes read, -1 is returned when the end of the stream is reached.

**Throws: IOException**

If an I/O error has occurred

3) `public int read(byte b[]) throws IOException`

Reads into an array of bytes. This method will block until some input is available.

**Parameters:**

b - the buffer into which the data is read

**Returns:**

the actual number of bytes read, -1 is returned when the end of the stream is reached.

**Throws: IOException**

If an I/O error has occurred

4) `public long skip(long n) throws IOException`

Skips n bytes of input.

**Parameters:**

n - the number of bytes to be skipped

**Returns:**

the actual number of bytes skipped.

**Throws: IOException**

If an I/O error has occurred.

5) `public int available() throws IOException`

Returns the number of bytes that can be read without blocking.

**Returns:**

the number of available bytes.

6) `public void close() throws IOException`

Closes the input stream. Must be called to release any resources associated with the stream.

**Throws: IOException**

If an I/O error has occurred.

## Some Common Methods of OutputStream:

### 1) write

`public abstract void write(int b) throws IOException`

Writes a byte. This method will block until the byte is actually written.

**Parameters:**

b - the byte

**Throws: IOException**

If an I/O error has occurred.

### 2) write

`public void write(byte b[]) throws IOException`

Writes an array of bytes. This method will block until the bytes are actually written.

**Parameters:**

b - the data to be written

**Throws: IOException**

If an I/O error has occurred.

### 3) write

`public void write(byte b[],`

```
        int off,  
        int len) throws IOException
```

Writes a sub array of bytes.

**Parameters:**

b - the data to be written

off - the start offset in the data

len - the number of bytes that are written

**Throws: IOException**

If an I/O error has occurred.

**4) flush**

```
public void flush() throws IOException
```

Flushes the stream. This will write any buffered output bytes.

**Throws: IOException**

If an I/O error has occurred.

**5) close**

```
public void close() throws IOException
```

Closes the stream. This method must be called to release any resources associated with the stream.

**Throws: IOException**

If an I/O error has occurred.

## FileInputStream

*FileInputStream is Use to Read Bytes From Specified File. Its Base Class Name is InputStream means this class extends all the methods of InputStream.*

### Constructors

**1) FileInputStream**

```
public FileInputStream(String name) throws FileNotFoundException
```

Creates an input file with the specified system dependent file name.

**Parameters:**

name - the system dependent file name

**Throws: IOException**

If the file is not found.

**2) FileInputStream**

```
public FileInputStream(File file) throws FileNotFoundException
```

Creates an input file from the specified File object.

**Parameters:**

file - the file to be opened for reading

**Throws: IOException**

If the file is not found.

**3) FileInputStream**

```
public FileInputStream(int fd) throws IOException
```

Creates an input file with the specified system dependent file descriptor.

**Parameters:**

fd - the system dependent file descriptor

**Throws: IOException**

If an I/O error has occurred

**Note: Methods of FileInputStream are same as InputStream**

## FileOutputStream

*FileOutputStream is Use to Write Bytes in Specified File. Its Base Class Name is OutputStream means this class extends all the methods of OutputStream.*

## **Constructors**

### **1) FileOutputStream**

```
public FileOutputStream(String name) throws IOException
```

Creates an output file with the specified system dependent file name.

#### **Parameters:**

name - the system dependent file name

#### **Throws: IOException**

If the file is not found.

### **2) FileOutputStream**

```
public FileOutputStream(int fd) throws IOException
```

Creates an output file with the specified system dependent file descriptor.

#### **Parameters:**

fd - the system dependent file descriptor

#### **Throws: IOException**

If an I/O error has occurred.

### **3) FileOutputStream**

```
public FileOutputStream(File file) throws IOException
```

Creates an output file with the specified File object.

#### **Parameters:**

file - the file to be opened for reading

#### **Throws: IOException**

If the file is not found.

**Note:** Methods of FileOutputStream are same as OutputStream

## **Example of InputStream/OutputStream/FileInputStream/FileOutputStream**

```
//Program to Copy File
import java.io.*;
class copyfile
{public static void main(String arg[])
{try{
DataInputStream X=new DataInputStream(System.in); // Create Object to Read Data From
Keyboard
System.out.println("Enter Source File:");
String src=X.readLine();
System.out.println("Enter Target File:");
String trgt=X.readLine();
InputStream fin=new FileInputStream(src);//open source file in read mode
OutputStream fout=new FileOutputStream(trgt);//open target file in write mode
int b;
while((b=fin.read())!=-1) //read a single byte from source file
{fout.write(b); //write in target file
}
fout.close();
fin.close();
System.out.println("File Copied....");
}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 constructor of the PrintStream class is written as:*

**PrintStream (java.io.OutputStream out);** //create a new print stream

The additional **print( )** and **Println( )** methods of this class give the same functionality as the method of standard output stream and follow the representations with newline.

The given example demonstrate the writing operation to a file using PrintStream class.

```
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("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.err.println ("Error in writing to file");
        }
    }
}
```

### **Output of the Program:**

```
C:\nisha>javac
PrintStreamDemo.java

C:\nisha>java
PrintStreamDemo
Write successfully
```

## Using a Random Access File

### **Introduction**

In this section, you will learn about the **RandomAccess File** class provided by **java.io** package. This is a class that allows you to read and write arbitrary bytes, text, and primitive Java data types from or to any specified location in a file. As the name suggests, random that means it is not sequential and the data can be read from and written to any specified position in the file.

Mostly, Users use the input stream or output stream in a sequential but Unlike the input and output streams **java.io. RandomAccessFile** is used for both reading and writing files A random access file treats with a large array of bytes stored in the file system and uses the **file pointer** to deal with the indexe of that array. This file pointer points the positions in the file where the reading or writing operation has to be done.

There are following methods has been used in this program:

**RandomAccessFile rand = new RandomAccessFile(file,"rw");**

The above code creates an instance of the **RandomAccessFile** class mentioning the mode in which file has to be opened. The constructor **RandomAccessFile( )** takes two arguments: First is the file name and another is the operation mode (read-write mode). We are opening the file into read and write mode ("rw").

There are following methods that are used in the given program shown as.

**rand.seek(file.length());**

This is the seek( ) method of the **RandomAccessFile** class has been used to jump the file pointer at the specified. Here, **file.length( )** returns the end of the file.

**rand.close();**

This is the close( ) method of the **RandomAccessFile** class has been used to close the created the instance of the **RandomAccessFile** class.

**writeBytes();**

This the **writeBytes( )** method which simply writes the content into the file. This method always append the file content with your specified text.

**readBytes()**

Read a Bytes from Files.

**Here is the code of the program :**

```
import java.io.*;

public class RandAccessFile{

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

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter File name : ");
        String str = in.readLine();

        File file = new File(str);
        if(!file.exists())
        {

            System.out.println("File does not exist.");
            System.exit(0);
        }

        try{
            //Open the file for both reading and writing
            RandomAccessFile rand = new RandomAccessFile(file,"rw");

            rand.seek(file.length()); //Seek to end of file
            rand.writeBytes("Softech India,"); //Write end of file

            rand.close();

            System.out.println("Write Successfully");
        }

        catch(IOException e)
        {

            System.out.println(e.getMessage());
        }
    }
}
```

**Output of the Program:**

```
C:\nisha>javac
RandAccessFile.java
```

```
C:\nisha>java
RandAccessFile
Enter File name :
Filterfile.txt
```

**Write Successfully**

The another program reads the characters from a file using the **readByte()** method.

```
import java.io.*;

public class ReadAccessFile{
    public static void main(String[] args) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter File name : ");
        String str = in.readLine();
        File file = new File(str);
        if(!file.exists()){
            System.out.println("File does not exist.");
            System.exit(0);
        }
        try{
            //Open the file for both reading and writing
            RandomAccessFile rand = new RandomAccessFile(file,"r");
            int i=(int)rand.length();
            System.out.println("Length: " + i);
            rand.seek(0); //Seek to start point of file
            for(int ct = 0; ct < i; ct++){
                byte b = rand.readByte(); //read byte from the file
                System.out.print((char)b); //convert byte into char
            }
            rand.close();
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }
    }
}
```

**Output of the Program:**

```
C:\nisha>java
ReadAccessFile
Enter File name :
Filterfile.txt
Length: 30
??this is a file
```

**Some OtherMethods RandomAccessFile**

**▪close()**

Closes the file.

**▪length()**

Returns the length of the file.

**▪read()**

Reads a byte of data.

**▪read(byte[], int, int)**

Reads a sub array as a sequence of bytes.

**▪read(byte[])**

Reads data into an array of bytes.

**▪readBoolean()**

Reads a boolean.

**▪readByte()**

Reads a byte.

**▪readChar()**

Reads a 16 bit char.

**▪readDouble()**

Reads a 64 bit double.

**▪readFloat()**

Reads a 32 bit float.

**▪readFully(byte[])**

Reads bytes, blocking until all bytes are read.

**▪readInt()**

Reads a 32 bit int.

**▪readLine()**

Reads a line terminated by a '\n' or EOF.

**▪readLong()**

Reads a 64 bit long.

**▪readShort()**

Reads 16 bit short.

**▪readUTF()**

Reads a UTF formatted String.

**▪readUnsignedByte()**

Reads an unsigned 8 bit byte.

**▪readUnsignedShort()**

Reads 16 bit short.

**▪seek(long)**

Sets the file pointer to the specified absolute position.

**▪skipBytes(int)**

**▪write(int)**

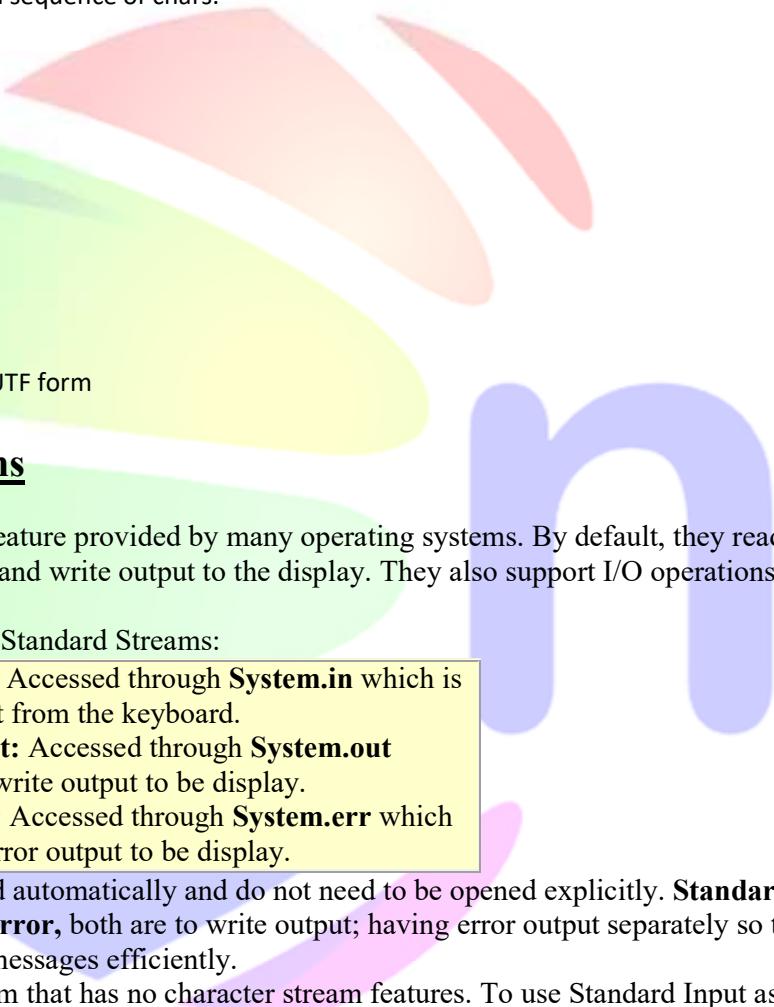
Writes a byte of data.

**▪write(byte[])**

Writes an array of bytes.

**▪write(byte[], int, int)**

Wrote a sub array of bytes.



- **writeBoolean(boolean)**  
Writes a boolean.
- **writeByte(int)**  
Writes a byte.
- **writeBytes(String)**  
Writes a String as a sequence of bytes.
- **writeChar(int)**  
Writes a character.
- **writeChars(String)**  
Writes a String as a sequence of chars.
- **writeDouble(double)**
- **writeFloat(float)**
- **writeInt(int)**  
Writes an integer.
- **writeLong(long)**  
Writes a long.
- **writeShort(int)**  
Writes a short.
- **writeUTF(String)**  
Writes a String in UTF form

## **Character Streams**

### **Standard Streams:**

Standard Streams are a feature provided by many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O operations on files.

Java also supports three Standard Streams:

- **Standard Input:** Accessed through **System.in** which is used to read input from the keyboard.
- **Standard Output:** Accessed through **System.out** which is used to write output to be display.
- **Standard Error:** Accessed through **System.err** which is used to write error output to be display.

These objects are defined automatically and do not need to be opened explicitly. **Standard Output** and **Standard Error**, both are to write output; having error output separately so that the user may read error messages efficiently.

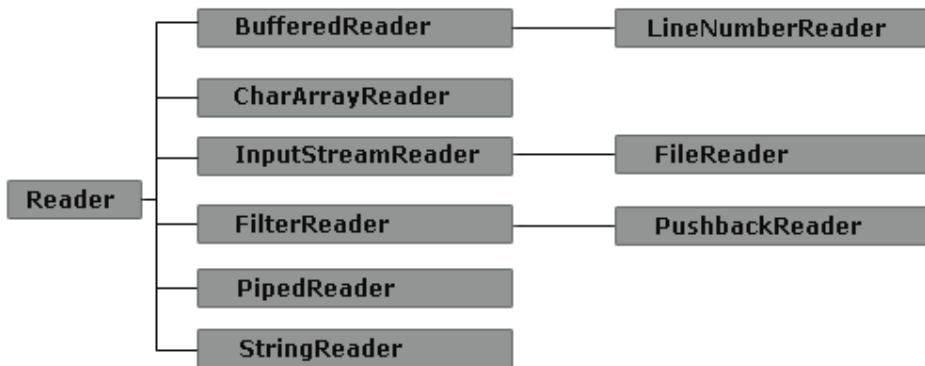
**System.in** is a byte stream that has no character stream features. To use Standard Input as a character stream, wrap **System.in** within the **InputStreamReader** as an argument.

**InputStreamReader inp = new InputStreamReader(system.in);**

### **Working with Reader classes:**

Java provides the standard I/O facilities for reading text from either the file or the keyboard on the command line. The **Reader** class is used for this purpose that is available in the **java.io** package. It acts as an abstract class for reading character streams. The only methods that a subclass must implement are **read(char[], int, int)** and **close()**. the Reader class is further categorized into the subclasses.

The following diagram shows a class-hierarchy of the **java.io.Reader** class.



However, most subclasses override some of the methods in order to provide higher efficiency, additional functionality, or both.

### **InputStreamReader:**

An InputStreamReader is a bridge from byte streams to character streams i.e. it reads **bytes** and decodes them into **Unicode characters** according to a particular platform. Thus, this class reads characters from a byte input stream. When you create an **InputStreamReader**, you specify an InputStream from which, the InputStreamReader reads the bytes.

The syntax of InputStreamReader is written as:

**InputStreamReader <variable\_name> = new InputStreamReader(system.in)**

### **BufferedReader :**

The **BufferedReader** class is the subclass of the **Reader** class. It reads character-input stream data from a memory area known as a **buffer** maintains state. The buffer size may be specified, or the default size may be used that is large enough for text reading purposes. BufferedReader converts an **unbuffered** stream into a **buffered** stream using the wrapping expression, where the unbuffered stream object is passed to the constructor for a buffered stream class.

For example the constructors of the BufferedReader class shown as:

**BufferedReader(Reader in)** : Creates a buffering character-input stream that uses a default-sized input buffer.

**BufferedReader(Reader in, int sz)**: Creates a buffering character-input stream that uses an input buffer of the specified size.

**BufferedReader** class provides some standard methods to perform specific reading operations shown in the table. All methods throws an **IOException**, if an I/O error occurs.

| Method                              | Return Type | Description                                                           |
|-------------------------------------|-------------|-----------------------------------------------------------------------|
| read( )                             | int         | Reads a single character                                              |
| read(char[] cbuf, int off, int len) | int         | Read characters into a portion of an array.                           |
| readLine( )                         | String      | Read a line of text. A line is considered to be terminated by ('\n'). |
| close()                             | void        | Closes the opened stream.                                             |

This program illustrates you how to use standard input stream to read the user input..

```

import java.io.*;

public class ReadStandardIO{

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

```

InputStreamReader inp = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(inp);

System.out.println("Enter text : ");

String str = in.readLine();

System.out.println("You entered String : ");
System.out.println(str);
}
}

```

### **Output of the Program:**

C:\nisha>javac  
ReadStandardIO.java

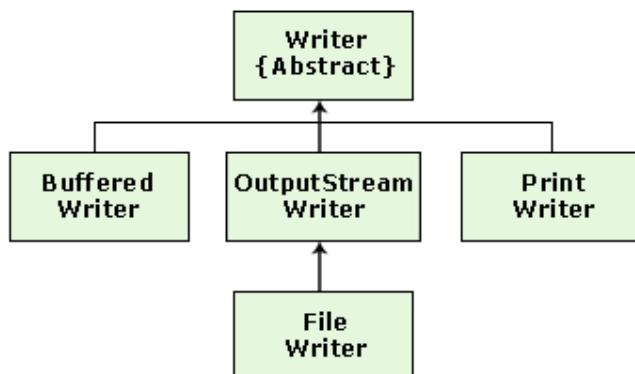
C:\nisha>java  
ReadStandardIO  
Enter text :  
this is an Input Stream  
You entered String :  
this is an Input Stream

C:\nisha>

## **Java writer**

Java **writer** class is an abstractly implemented idea for writing text files using character streams. It is defined inside **java.io** package. This class is the root of all the classes that write character output streams. In short, **writer** class provides foundation for the classes that write character output streams. This java writer hierarchy contains many classes such as **PrintWriter**, **BufferedWriter**, **FileWriter**, **OutputStreamWriter** and many more.

Diagram below demonstrates a subset of the **Writer** hierarchy. Diagram demonstrates the classes that are used to create an **OutputStream** environment of characters for writing text files.



## **Methods Of Writer Class**

### **1) write()**

Method writes characters in to the stream. This method has several constructor forms as mentioned below.

**write(char[] cbuf)** :- With this constructor form method writes an array of characters.

**write(char[] cbuf, int off, int len)** :- With this constructor form method writes a portion or part of an array of characters.

**write(int c)** :- With this only a single character can written.

**write(String str)** :- With this a string value can be written in to the stream.

**write(String str, int off, int len)**:- With this a subsequence or portion of string value is written to the stream.

### **2) close()**

This method is used to close the underlying stream.

### **3) flush()**

Method flushes the stream. Flushing means flowing the data from the stream without closing it.

**PrintWriter** :- Class PrintWriter extends Writer class. Class inherits all the methods of **PrintStream** and is used to write text output streams. The texts written by class are the formatted representations of other **Writer** class objects. 'formatted representations' means the data is already taken and buffered. Class takes the **Writer** objects as argument. Class does not show any IOException, but its constructors can generate this exception. In the class with **checkError()**, programmer can check for any generated errors.

**FileWriter** :- Class **FileWriter** has public access and inherits **OutputStreamWriter class**. Class usually used for connecting a character output stream to file. Class with its inherited methods can also write character files. Class can only write streams of characters and not the raw bytes. For writing raw bytes **FileOutputStream** class is used.

**BufferedWriter**:- A subclass inside abstract writer class that creates a buffer for the character output stream for its better processing. The buffer created by the class is a temporary memory where the characters written into character output stream is buffered. The buffer created has its default size which is large enough for most purposes but this size can be specified.

In order to provide the buffering functionality in the filtered stream. This class is layered with other writer subclasses to make filtered stream required.

Syntax for creating class object :- **BufferedWriter br = new BufferedWriter();**

Syntax for creating a filtered stream:- **PrintWriter out = new PrintWriter (new BufferedWriter (new FileWriter ( "rose.txt" )));**

If no buffer is set for the character out stream, then when ever the print() method is invoked, the characters will directly convert in to bytes and will be instantly written to the file, and that would be very disorganized.

**//Program to Write Character in File using PrintWriter/BuffredWiter**

```
import java.io.*;
class test
{public static void main(String arg[])
{try{
FileWriter fw=new FileWriter("softech.txt");//Open in Write Mode
BufferedWriter bw=new BufferedWriter (fw);// Create Buffer To Write Character
PrintWriter out = new PrintWriter (bw);//Create PrintWriter Object
```

```
out.println("Softech. India"); //Write Data In File  
out.close();  
System.out.println("Write Successfully...");  
}catch(Exception e)  
{System.out.println(e);  
}}}
```

```
//Program to Display Contents On Out put Device Using PrintWriteter  
import java.io.*;  
class test  
{public static void main(String arg[])  
{try{  
OutputStreamWriter out=new OutputStreamWriter(System.out);  
PrintWriter p=new PrintWriter(out);  
p. p.close();  
}catch(Exception e)  
{System.out.println(e);}}}
```

## Serializing an Object in Java

### Introduction

**Serialization** is the process of saving an object in a storage medium (such as a file, or a memory buffer) or to transmit it over a network connection in binary form. The serialized objects are JVM independent and can be re-serialized by any JVM. In this case the "in memory" java objects state are converted into a byte stream. This type of the file can not be understood by the user. It is a special types of object i.e. reused by the JVM (Java Virtual Machine). This process of serializing an object is also called **deflating** or **marshalling** an object.

The given example shows the implementation of serialization to an object. This program takes a file name that is machine understandable and then serialize the object. The object to be serialized must implement **java.io.Serializable** class.

Default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields.

class **ObjectOutputStream** extends **java.io.OutputStream** implements **ObjectOutput**, **ObjectOutput** interface extends the **DataOutput** interface and adds methods for serializing objects and writing bytes to the file. The **ObjectOutputStream** extends **java.io.OutputStream** and implements **ObjectOutput** interface. It serializes objects, arrays, and other values to a stream. Thus the constructor of **ObjectOutputStream** is written as:

**ObjectOutput ObjOut = new ObjectOutputStream(new FileOutputStream(f));**

Above code has been used to create the instance of the **ObjectOutput** class with the **ObjectOutputStream( )** constructor which takes the instance of the **FileOuputStream** as a parameter.

The **ObjectOutput** interface is used by implementing the **ObjectOutputStream** class. The **ObjectOutputStream** is constructed to serialize the object.

### **writeObject():**

Method writeObject() writes an object to the given stream.

### **Here is the code of program:**

```
import java.io.*;
public class SerializingObject{

    public static void main(String[] args) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Please enter File name : ");
        String file = in.readLine();
        System.out.print("Enter extention : ");
        String ext = in.readLine();
        String filename = file + "." + ext;
        File f = new File(filename);
        try{
            ObjectOutputStream ObjOut = new ObjectOutputStream(new FileOutputStream(f));
            ObjOut.writeObject(f);
            ObjOut.close();
            System.out.println("Serializing an Object Creation completed successfully.");
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }
    }
}
```

### **Output of the Program:**

```
C:\nisha>javac
SerializingObject.java

C:\nisha>java
SerializingObject
Please enter File name :
Filterfile
Enter extention : txt
Serializing an Object Creation
is completely Successfully.
```

# Deserializing an Object in java

## Introduction

The opposite operation of the serialization is called **deserialization** i.e. to extract the data from a series of bytes is known as **deserialization** which is also called **inflating** or **unmarshalling**.

The given program shows how to read any data or contents from the serialized object or file. It takes a file name and then converts into java object. If any exception occurs during reading the serialized file, it is caught in the catch block.

**ObjectInputStream** extends **java.io.InputStream** and implements **ObjectInput** interface. It deserializes objects, arrays, and other values from an input stream. Thus the constructor of **ObjectInputStream** is written as:

```
ObjectInputStream obj = new ObjectInputStream(new FileInputStream(f));
```

Above code of the program creates the instance of the **ObjectInputStream** class to deserialize that file which had been serialized by the **ObjectOutputStream** class. The above code creates the instance using the instance of the **FileInputStream** class which holds the specified file object which has to be deserialized because the **ObjectInputStream()** constructor needs the input stream.

**readObject()::**

Method **readObject()** reads the object and restore the state of the object. This is the method of the **ObjectInputStream** class that helps to traverse the object.

**Here is a code of program :**

```
import java.io.*;
public class DeserializingObject{

    public static void main(String[] args) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter File name : ");

        String file = in.readLine();

        System.out.print("Enter extention : ");

        String ext = in.readLine();

        String filename = file + "." + ext;

        File f = new File(filename);

        try{

            ObjectInputStream obj = new ObjectInputStream(new FileInputStream(f));
            System.out.println("The text : "+ obj.readObject());
            obj.close();
            System.out.println("Deserializing Operation Completely Successfully.");
        }

        catch(ClassNotFoundException e){

            System.out.println(e.getMessage());
        }
    }
}
```

```

    }

    catch(FileNotFoundException fe) {
        System.out.println("File not found ");
    }
}
}

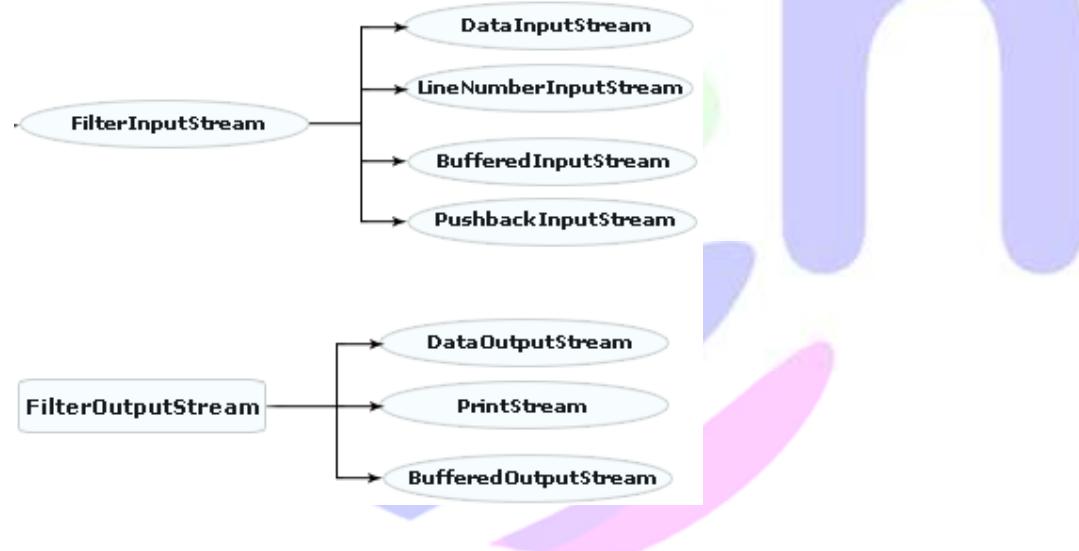
```

## Some Other Useful IO Stream Classes

### Overview of I/O Data Streams

As mentioned earlier, **Filter streams** are special streams which filter input and output bytes and manipulate data reading from an underlying stream. Apart from this, it allows the user to make a chain using multiple input stream so that, the operations that are to be applied on this chain, may create a combine effects on several filters. By using these streams, there is no need to convert the data from **byte** to **char** while writing to a file. These are the more powerful streams than the other streams of Java.

The class hierarchy of the Filter I/O streams derived from I/O streams can be shown as:



In this section we will learn about the Data I/O streams derived from the Filter I/O streams.

### **DataStreams:**

Data streams are filtered streams that perform binary I/O operation on primitive data type values ( boolean, char, byte, short, int, long, etc.) as well as on String values. If you need to work with data that is not represented as bytes or characters then you can use Data Streams. These streams filter an existing byte stream so that each primitive data types can be read from or written to the stream directly.

These Data Streams are as follows:

- **DataInputStream**
- **DataOutputStream**

#### **DataInputStream:**

This is a class derived from **FilterInputStream** class that allows you to read binary represented data of Java primitive data types from an underlying input stream in a machine-independent way. It reads only Java primitive data types and doesn't read the object values. A data input stream is created with the constructor of **DataInputStream** class. The specified argument that is to be filtered within the constructor should be an existing input stream (such as a **buffered input stream** or a **file input stream**).

The constructor of DataInputStream is written as:

**DataInputStream** (java.io.InputStream in);

The **read( )** method is used to read the data according to its types. For example, the **readInt( )** method reads the **int** type of value while the **readFloat()** method reads the fraction value.

The **readLine( )** Methods reads the string per line from a file.

#### **DataOutputStream:**

This is a class derived from **FilterOutputStream** class that allows you to write binary represented data of Java primitive data types reading from an underlying output stream in a machine-independent way. It writes only Java primitive data types and doesn't write the object values. A data output stream is created with the constructor of **DataOutputStream** class. The specified argument that is to be filtered within the constructor should be an existing output stream (such as a **buffered output stream** or a **file output stream**).

The constructor of DataOutputStream is written as:

**DataOutputStream** (java.io.OutputStream out);

The **write( )** method is used to write the data according to its types. For example, the **writeInt( )** method writes the **int** type of value while the **writeFloat()** method writes the fraction value. The **writeUTF( )** method writes the string per line to a file.

Lets see an example that shows the implementation of reading and writing operations through the Data I/O streams.

```
import java.io.*;

class ReadWriteData
{
    public static void main(String args[])
    {
        int ch=0;
        int[] numbers = { 12, 8, 13, 29, 50 };
        try
        {

            System.out.println("1. write Data");
            System.out.println("2. Read Data");
            System.out.println("Enter your choice ");
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        };
        ch=Integer.parseInt(br.readLine());
        switch(ch){
            case 1:
                FileOutputStream fos = new FileOutputStream("datafile.txt");
                BufferedOutputStream bos = new BufferedOutputStream(fos);
                for(int i=0;i<numbers.length;i++)
                {
                    bos.write(numbers[i]);
                }
                bos.close();
        }
    }
}
```

```

DataOutputStream out =new DataOutputStream (bos);
for (int i = 0; i < numbers.length; i ++) {
    out.writeInt(numbers[i]);
}
System.out.print("write successfully");
out.close();
case 2:
    FileInputStream fis = new FileInputStream("datafile.txt");
    BufferedInputStream bis=new BufferedInputStream(fis);
    DataInputStream in =new DataInputStream (bis);
    while (true){
        System.out.print(in.readInt());
    }

    default:
        System.out.println("Invalid choice");
    }
}
catch (Exception e) {
    System.err.println("Error in read/write data to a file: " + e);
} }}}

```

### **Output of the Program:**

```

C:\nisha>javac
ReadWriteData.java
C:\nisha>java
ReadWriteData
1. write Data
2. Read Data
Enter your choice
1
write
successfully128132950

```

This program uses both **DataInputStream** and **DataOutputStream** to read and write data to the specified file respectively. If the user enters the choice as 1 then the specified data is written to a file through the object of **DataOutputStream** class using the **writeInt( )** method. On the other hand, if the user enters the choice as 2 then the written data is read from the file through the object of **DataInputStream** class using the **ReadInt( )**.

Here, another output is shown when the user enters the choice as 2.

```

C:\nisha>javac
ReadWriteData.java
C:\nisha>java
ReadWriteData
1. write Data
2. Read Data
Enter your choice
2
128132950
C:\nisha>

```

## **Working With File**

### **File class**

The **File** class deals with the machine dependent files in a machine-independent manner i.e. it is easier to write platform-independent code that examines and manipulates files using the **File** class. This class is available in the **java.lang** package.

The **java.io.File** is the central class that works with files and directories. The **instance** of this class represents the name of a file or directory on the host file system.

When a File object is created, the system doesn't check to the existence of a corresponding file/directory. If the file exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, reading from or writing to it.

The **constructors** of the File class are shown in the table:

| <b>Constructor</b>                | <b>Description</b>                                                           |
|-----------------------------------|------------------------------------------------------------------------------|
| <code>File(path)</code>           | Create File object for default directory (usually where program is located). |
| <code>File(dirpath, fname)</code> | Create File object for directory path given as string.                       |
| <code>File(dir, fname)</code>     | Create File object for directory.                                            |

Thus the statement can be written as:

```
File f = new File("<filename>");
```

The methods that are used with the file object to get the attribute of a corresponding file shown in the table.

| <b>Method</b>                  | <b>Description</b>                                                                                              |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>f.exists()</code>        | Returns true if file exists.                                                                                    |
| <code>f.isFile()</code>        | Returns true if this is a normal file.                                                                          |
| <code>f.isDirectory()</code>   | true if "f" is a directory.                                                                                     |
| <code>f.getName()</code>       | Returns name of the file or directory.                                                                          |
| <code>f.isHidden()</code>      | Returns true if file is hidden.                                                                                 |
| <code>f.lastModified()</code>  | Returns time of last modification.                                                                              |
| <code>f.length()</code>        | Returns number of bytes in file.                                                                                |
| <code>f.getPath()</code>       | path name.                                                                                                      |
| <code>f.delete()</code>        | Deletes the file.                                                                                               |
| <code>f.renameTo(f2)</code>    | Renames f to File f2. Returns true if successful.                                                               |
| <code>f.createNewFile()</code> | Creates a file and may throw IOException.                                                                       |
| <code>String[] list()</code>   | Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname |
| <code>boolean mkdir()</code>   | Creates the directory named by this abstract pathname.                                                          |

### **Java Directory - Directory and File Listing Example in Java**

This example illustrates how to list files and folders present in the specified directory. This topic is related to the I/O (input/output) of `java.io` package.

In this example we are using `File` class of `java.io` package. The `File` class is an abstract representation of file and directory pathnames. This class is an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, "/" for the UNIX root directory, or "\\\" for a Win32 UNC pathname, and
2. A sequence of zero or more string *names*.

### Explanation

This program lists the files of the specified directory. We will be declaring a function called `dirlist` which lists the contents present in the specified directory.

`dirlist(String fname)`

The function `dirlist(String fname)` takes directory name as parameter. The function creates a new `File` instance for the directory name passed as parameter

`File dir = new File(fname);`

and retrieves the list of all the files and folders present in the directory by calling `list()` method on it.

`String[] chld = dir.list();`

Then it prints the name of files and folders present in the directory.

### Code of the Program :

```
import java.io.*;  
  
public class DirListing{  
    private static void dirlist(String fname){  
        File dir = new File(fname);  
        String[] chld = dir.list();  
        if(chld == null){  
            System.out.println("Specified directory does not exist or is not a directory.");  
            System.exit(0);  
        }else{  
            for(int i = 0; i < chld.length; i++){  
                String fileName = chld[i];  
                System.out.println(fileName);  
            }  
        }  
    }  
    public static void main(String[] args){  
        switch(args.length){  
            case 0: System.out.println("Directory has not mentioned.");  
                    System.exit(0);  
            case 1: dirlist(args[0]);  
                    System.exit(0);  
            default : System.out.println("Multiple files are not allowed.");  
                    System.exit(0);  
        }  
    }  
}
```

### Java Create Directory

In the section, you will learn how a directory or subdirectories are created. This program also explains the process of creating all non-existent ancestor directories automatically. We will use the `File` class to create the directory.

## File

The File class an abstract representation of file and directory pathnames. File class is used to interact with the files system.

```
import java.io.*;
class CreateDirectory
{
    public static void main(String args[])
    {
        try{
            String strDirectoy ="test";
            String strManyDirectories="dir1/dir2/dir3";

            // Create one directory
            boolean success = (new File(strDirectoy)).mkdir();
            if (success) {
                System.out.println("Directory: " + strDirectoy + " created");
            }

            // Create multiple directories
            success = (new File(strManyDirectories)).mkdirs();
            if (success) {
                System.out.println("Directories: " + strManyDirectories + " created")
            }
        }catch (Exception e){//Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

### Output of the program:

```
C:\nisha>javac
.CreateDirectory.java
```

```
C:\nisha>java
.CreateDirectory
Directory: test created
Directories:
dir1/dir2/dir3 created
```

```
C:\nisha>
```

This program takes inputs to create a directory "test" and subdirectories "dir1\dir2\dir3". the **mkdir()** method is used to create a single directory while the **mkdirs()** method is used to create multiple subdirectories.



# Java Database Connectivity

## Java Database Connectivity.

**JDBC** is Java application programming interface that allows the Java programmers to access database management system from Java code. It was developed by **JavaSoft**, a subsidiary of **Sun Microsystems**.

### Definition

**Java Database Connectivity** in short called as JDBC. It is a java API which enables the java programs to execute SQL statements. It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.

JDBC provides methods for querying and updating the data in Relational Database Management system such as SQL, Oracle etc.

JDBC connections which supports creating and executing statements such as SQL INSERT, UPDATE and DELETE. Driver Manager is the backbone of the jdbc architecture.

Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not depended upon any language.

In short JDBC helps the programmers to write java applications that manage these three programming activities:

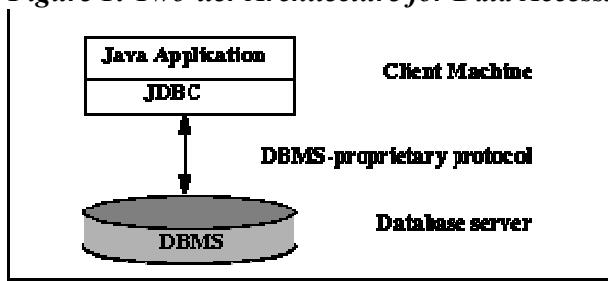
- 1. It helps us to connect to a data source, like a database.*
- 2. It helps us in sending queries and updating statements to the database and*
- 3. Retrieving and processing the results received from the database in terms of answering to your query.*

## JDBC Architecture

### 1) Two-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

*Figure 1: Two-tier Architecture for Data Access.*

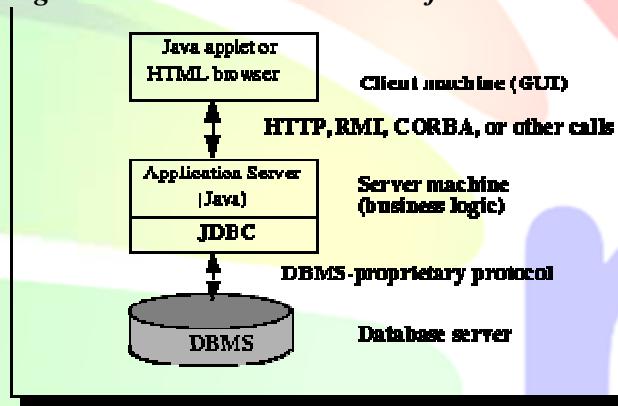


In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

## 2) Three-tier Processing Models

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.

*Figure 2: Three-tier Architecture for Data Access.*



With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

### Connectivity Models Of JDBC

#### Type 1 Driver - JDBC-ODBC bridge

The JDBC type 1 driver, also known as the JDBC-ODBC bridge, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.

#### Functions

- Translates query obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver.
- Sun provides a JDBC-ODBC Bridge driver. `sun.jdbc.odbc.JdbcOdbcDriver`. This driver is native code and not Java, and is closed source.
- Client -> JDBC Driver -> ODBC Driver -> Database
- There is some overhead associated with the translation work to go from JDBC to ODBC.

#### Advantages

- Almost any database for which ODBC driver is installed, can be accessed.
- A type 1 driver is easy to install.

## Disadvantages

- Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native db connectivity interface.
- The ODBC driver needs to be installed on the client machine.
- Considering the client-side software needed, this might not be suitable for applets.
- Will not be suitable for Java applets.
- This driver will work only on Windows since ODBC is the Windows based application.

## Type 2 Driver - Native-API Driver specification

The JDBC type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

The type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the final database calls. However the type 2 driver provides more functionality and better performance than the type 1 driver as it does not have the overhead of the additional ODBC function calls.

## Advantages

- Better performance than Type 1 Driver (JDBC-ODBC bridge).

## Disadvantages

- The vendor client library needs to be installed on the client machine.
- Cannot be used in web-based application due to the client side software needed.
- Not all databases have a client side library
- This driver is platform dependent

## Type 3 Driver - Network-Protocol Driver

The JDBC type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

## Functions

- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- Thus the client driver to middleware communication is database independent.
- Client -> JDBC Driver -> Middleware-Net Server -> Any Database

## Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the vendor db library on the client machine. Also the client to middleware need not be changed for a new database.
- The Middleware Server (which can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing etc.
- Eg. for the above include jdbc driver features in Weblogic.
  - Can be used in internet since there is no client side software needed.
  - At client side a single driver can handle any database. (It works provided the middleware supports that database!)

## Disadvantages

- Requires database-specific coding to be done in the middle tier.
- An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware services described above.

## Type 4 Driver - Native-Protocol Driver

The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into the vendor-specific database protocol.

The type 4 driver is written completely in Java and is hence platform independent. It is installed inside the Java Virtual Machine of the client. It provides better performance over the type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 3 drivers, it does not need associated software to work. As the database protocol is vendor-specific, separate drivers, usually vendor-supplied, need to be used to connect to the database.

## Functions

- Type 4 drivers are entirely written in Java that communicate directly with a vendor's database, usually through socket connections. No translation or middleware layers are required, improving performance.
- The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
- Completely implemented in Java to achieve platform independence.
- e.g. include the widely used Oracle thin driver - oracle.jdbc.driver.OracleDriver which connect to jdbc:oracle:thin URL format.
- Client Machine -> Native protocol JDBC Driver -> Database server

## Advantages

- These drivers don't translate the requests into an intermediary format (such as ODBC), nor do they need a middleware layer to service requests. Thus the performance may be considerably improved.
- All aspects of the application to database connection can be managed within the JVM; this can facilitate easier debugging.

## Disadvantage

- At client side, a separate driver is needed for each database.

# JDBC API

JDBC is an API specification developed by **Sun Microsystems** that defines a uniform interface for accessing various relational databases. JDBC is a core part of the Java platform and is included in the standard JDK distribution.

The primary function of the JDBC API is to provide a means for the developer to issue SQL statements and process the results in a consistent, database-independent manner.

JDBC provides rich, object-oriented access to databases by defining classes and interfaces that represent objects such as:

1. Database connections
2. SQL statements
3. Result Set
4. Database metadata
5. Prepared statements
6. Callable statements
7. Database drivers
8. Driver manager

## Connection interface

A **Connection** object represents a connection with a database. When we connect to a database by using connection method, we create a Connection Object, which represents the connection to the database. An application may have one or more than one connections with a single database or many connections with the different databases also.

java.sql package  
public interface **Connection**

| <b><i>Connection interface methods</i></b> |                                                                                                                                             |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Method return value type</b>            | <b>Method</b>                                                                                                                               |
| void                                       | <b>close()</b><br>Releases a Connection's database and JDBC resources immediately instead of waiting for them to be automatically released. |
| Statement                                  | <b>createStatement()</b><br>Creates a Statement object for sending SQL statements to the database.                                          |
| DatabaseMetaData                           | <b>getMetaData()</b><br>Gets the metadata regarding this Connection's database.                                                             |
| CallableStatement                          | <b>prepareCall(String sql)</b><br>Creates a CallableStatement object for calling database stored procedures.                                |
| PreparedStatement                          | <b>prepareStatement(String sql)</b><br>Creates a PreparedStatement object for sending parameterized SQL statements to the database.         |

## Statement interface

The Statement interface creates an object that is used to execute a static SQL statement and obtain the results produced by it.

java.sql package

public interface **Statement**

### *Statement interface methods*

| Method return value type | Method                                                                                                                                                                  |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void                     | <code>close()</code><br>Releases this Statement object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. |
| boolean                  | <code>execute(String sql)</code><br>Executes an SQL statement that might return multiple results.                                                                       |
| int []                   | <code>executeBatch()</code> JDBC 2.0 Submits a batch of commands to the database for execution.                                                                         |
| ResultSet                | <code>executeQuery(String sql)</code><br>Executes an SQL statement that returns a single ResultSet object.                                                              |
| int                      | <code>executeUpdate(String sql)</code><br>Executes an SQL INSERT, UPDATE, or DELETE statement.                                                                          |

## ResultSet interface

The ResultSet interface provides access to a table of data. A ResultSet object is usually generated by executing a Statement.

A ResultSet maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row. The `next()` method moves the cursor to the next row. The `getXXX` methods retrieve column values for the current row. You can retrieve values using either the index number of the column or the name of the column. In general, using the column index is more efficient. Columns are numbered from one. The JDBC driver converts the underlying data to the Java™ type specified in the getter method and returns a suitable Java value.

java.sql package

public interface **ResultSet**

### *ResultSet interface methods*

| Method return value type | Method                                                                                                        |
|--------------------------|---------------------------------------------------------------------------------------------------------------|
| boolean                  | <code>absolute(int row)</code> JDBC 2.0.<br>Moves the cursor to the given row number in the result set.       |
| void                     | <code>afterLast()</code> JDBC 2.0.<br>Moves the cursor to the end of the result set, just after the last row. |

| <b><i>ResultSet interface methods</i></b> |                                                                                                                                                                                                                                              |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Method return value type</b>           | <b>Method</b>                                                                                                                                                                                                                                |
| void                                      | beforeFirst() JDBC 2.0.<br>Moves the cursor to the front of the result set, just before the first row.                                                                                                                                       |
| void                                      | close()<br>Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.                                                                                   |
| boolean                                   | first() JDBC 2.0.<br>Moves the cursor to the first row in the result set.                                                                                                                                                                    |
| boolean                                   | getBoolean(int columnIndex)<br>Gets the value of a column in the current row as a Java boolean. The driver first gets the value of the column as a Java short. If the value is equal to 1, true is returned. Otherwise, false is returned.   |
| boolean                                   | getBoolean(String columnName)<br>Gets the value of a column in the current row as a Java boolean. The driver first gets the value of the column as a Java short. If the value is equal to 1, true is returned. Otherwise, false is returned. |
| byte                                      | getByte(int columnIndex)<br>Gets the value of the designated column in the current row of this ResultSet object as a byte in the Java programming language.                                                                                  |
| byte                                      | getByte(String columnName)<br>Gets the value of the designated column in the current row of this ResultSet object as a byte in the Java programming language.                                                                                |
| byte []                                   | getBytes(int columnIndex)<br>Gets the value of the designated column in the current row of this ResultSet object as a byte array in the Java programming language.                                                                           |
| byte []                                   | getBytes(String columnName)<br>Gets the value of the designated column in the current row of this ResultSet object as a byte array in the Java programming language.                                                                         |
| Date                                      | getDate(int columnIndex)<br>Gets the value of the designated column in the current row of this ResultSet object as a java.sql.Date object in the Java programming language.                                                                  |
| Date                                      | getDate(int columnIndex, Calendar cal)<br>Returns the value of the designated column in the current row of this ResultSet object as a java.sql.Date object in the Java programming language.                                                 |
| Date                                      | getDate(String columnName)                                                                                                                                                                                                                   |

| <b><i>ResultSet interface methods</i></b> |                                                                                                                                                                              |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Method return value type</b>           | <b>Method</b>                                                                                                                                                                |
|                                           | Gets the value of the designated column in the current row of this ResultSet object as a java.sql.Date object in the Java programming language.                              |
| double                                    | <code>getDouble(int columnIndex)</code><br>Gets the value of a column in the current row as a Java double.                                                                   |
| double                                    | <code>getDouble(String columnName)</code><br>Gets the value of a column in the current row as a Java double.                                                                 |
| float                                     | <code>getFloat(int columnIndex)</code><br>Gets the value of a column in the current row as a Java float.                                                                     |
| float                                     | <code>getFloat(String columnName)</code><br>Gets the value of a column in the current row as a Java float.                                                                   |
| int                                       | <code>getInt(int columnIndex)</code><br>Gets the value of the designated column in the current row of this ResultSet object as an integer in the Java programming language.  |
| int                                       | <code>getInt(String columnName)</code> Gets the value of the designated column in the current row of this ResultSet object as an integer in the Java programming language.   |
| long                                      | <code>getLong(int columnIndex)</code><br>Gets the value of a column in the current row as a Java long.                                                                       |
| long                                      | <code>getLong(String columnName)</code><br>Gets the value of a column in the current row as a Java long.                                                                     |
| ResultSetMetaData                         | <code>getMetaData()</code><br>Retrieves the number, types, and properties of this ResultSet object's columns.                                                                |
| Object                                    | <code>getObject(int columnIndex)</code><br>Gets the value of a column in the current row as a Java object.                                                                   |
| Object                                    | <code>getObject(String columnName)</code><br>Gets the value of a column in the current row as a Java object.                                                                 |
| int                                       | <code>getRow() JDBC 2.0.</code><br>Retrieves the current row number.                                                                                                         |
| short                                     | <code>getShort(int columnIndex)</code><br>Gets the value of the designated column in the current row of this ResultSet object as a short in the Java programming language.   |
| short                                     | <code>getShort(String columnName)</code><br>Gets the value of the designated column in the current row of this ResultSet object as a short in the Java programming language. |

| <b><i>ResultSet interface methods</i></b> |                                                                                                                                                                                            |
|-------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Method return value type</b>           | <b>Method</b>                                                                                                                                                                              |
| String                                    | <code>getString(int columnIndex)</code><br>Gets the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.               |
| String                                    | <code>getString(String columnName)</code><br>Gets the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.             |
| Time                                      | <code>getTime(int columnIndex)</code><br>Gets the value of the designated column in the current row of this ResultSet object as a java.sql.Time object in the Java programming language.   |
| Time                                      | <code>getTime(String columnName)</code><br>Gets the value of the designated column in the current row of this ResultSet object as a java.sql.Time object in the Java programming language. |
| boolean                                   | <code>isAfterLast() JDBC 2.0.</code><br>Indicates whether the cursor is after the last row in the result set.                                                                              |
| boolean                                   | <code>isBeforeFirst() JDBC 2.0.</code><br>Indicates whether the cursor is before the first row in the result set.                                                                          |
| boolean                                   | <code>isFirst() JDBC 2.0.</code><br>Indicates whether the cursor is on the first row of the result set.                                                                                    |
| boolean                                   | <code>isLast() JDBC 2.0.</code><br>Indicates whether the cursor is on the last row of the result set. This method is not supported for result sets with type TYPE_FORWARD_ONLY.            |
| boolean                                   | <code>last() JDBC 2.0.</code><br>Moves the cursor to the last row in the result set.                                                                                                       |
| boolean                                   | <code>next()</code><br>Moves the cursor down one row from its current position.                                                                                                            |
| boolean                                   | <code>previous() JDBC 2.0.</code><br>Moves the cursor to the previous row in the result set.                                                                                               |

## **ResultSetMetaData interface**

The ResultSetMetaData interface creates an object that can be used to find out about the types and properties of the columns in a ResultSet.

java.sql package

| <i>ResultSetMetaData interface methods</i> |                                                                                                        |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <b>Method return value type</b>            | <b>Method</b>                                                                                          |
| int                                        | getColumnName() Returns the number of columns in this ResultSet object.                                |
| String                                     | getColumnName (int column) Gets the designated column's name.                                          |
| int                                        | getColumnType (int column) Gets the designated column's SQL type.                                      |
| String                                     | getColumnTypeName(int column) Retrieves a column's database-specific type name.                        |
| int                                        | getPrecision (int column) Gets the designated column's number of decimal digits.                       |
| int                                        | getScale (int column) Gets the designated column's number of digits to the right of the decimal point. |
| int                                        | isNullable (int column) Indicates the nullability of values in the designated column.                  |
| boolean                                    | isWritable(int column) Indicates whether it is possible for a write on the column to succeed.          |

## **PreparedStatement interface**

The PreparedStatement interface creates an object that is used to create parameterized Query. One can define Parameter with thw help of ? Symbol

java.sql package

public interface **PreparedStatement**

extends Statement

| <i>PreparedStatement interface methods</i> |                                                  |
|--------------------------------------------|--------------------------------------------------|
| <b>Method return value type</b>            | <b>Method</b>                                    |
| boolean                                    | execute()<br>Executes any kind of SQL statement. |

| <b><i>PreparedStatement interface methods</i></b> |                                                                                                                                                                                                                                                                                                  |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Method return value type</b>                   | <b>Method</b>                                                                                                                                                                                                                                                                                    |
| ResultSet                                         | <b>executeQuery()</b><br>Executes the SQL query in this PreparedStatement object and returns the result set generated by the query.                                                                                                                                                              |
| Int                                               | <b>executeUpdate()</b><br>Executes the SQL INSERT, UPDATE or DELETE statement in this PreparedStatement object.                                                                                                                                                                                  |
| Void                                              | <b>setBoolean (int parameterIndex, boolean x)</b><br>Sets the designated parameter to a Java™ boolean value. The DB2 Everyplace JDBC driver converts this to an SQL SMALLINT value when it sends it to the database.                                                                             |
| void                                              | <b>setBytes(int parameterIndex, byte[ ] x)</b><br>Sets the designated parameter to a Java array of bytes.                                                                                                                                                                                        |
| void                                              | <b>setDate(int parameterIndex, Date x)</b><br>Sets the designated parameter to a java.sql.Date value.                                                                                                                                                                                            |
| void                                              | <b>setDouble(int parameterIndex, double x)</b><br>Sets the designated parameter to a Java double value. The DB2 Everyplace JDBC driver converts this to an SQL DECIMAL value when it sends it to the database.                                                                                   |
| void                                              | <b>setFloat(int parameterIndex, float x)</b><br>Sets the designated parameter to a Java float value. When a BigDecimal is converted to float, if the BigDecimal is too large to represent as a float, it will be converted to FLOAT.NEGATIVE_INFINITY or FLOAT.POSITIVE_INFINITY as appropriate. |
| void                                              | <b>setInt (int parameterIndex, int x)</b><br>Sets the designated parameter to a Java int value.                                                                                                                                                                                                  |
| void                                              | <b>setLong(int parameterIndex, long x)</b><br>Sets the designated parameter to a Java long value.                                                                                                                                                                                                |
| void                                              | <b>setNull (int parameterIndex, int sqlType)</b><br>Sets the designated parameter to SQL NULL.                                                                                                                                                                                                   |
| void                                              | <b>setShort (int parameterIndex, short x)</b><br>Sets the designated parameter to a Java short value.                                                                                                                                                                                            |
| void                                              | <b>setString (int parameterIndex, String x)</b><br>Sets the designated parameter to a Java String value.                                                                                                                                                                                         |
| void                                              | <b>setTime (int parameterIndex, Time x)</b><br>Sets the designated parameter to a java.sql.Time value.                                                                                                                                                                                           |

## CallableStatement interface

The interface used to execute remote SQL stored procedures. The result parameter must be registered as an OUT parameter. The other parameters can be used for input, output or both. Parameters are referred to sequentially, by number. The first parameter is 1.

```
call <procedure-name> (?, ?, ...)
```

A CallableStatement can return one ResultSet.

java.sql package

```
public interface CallableStatement  
extends PreparedStatement
```

| CallableStatement interface methods |                                                                                                                                                   |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| Method return value type            | Method                                                                                                                                            |
| Date                                | getDate(int parameterIndex) Gets the value of a JDBC DATE parameter as a java.sql.Date object.                                                    |
| int                                 | getInt(int parameterIndex) Gets the value of a JDBC INTEGER parameter as an int in the Java programming language.                                 |
| Object                              | getObject(int parameterIndex) Gets the value of a parameter as an object in the Java programming language.                                        |
| short                               | getShort(int parameterIndex) Gets the value of a JDBC SMALLINT parameter as a short in the Java programming language.                             |
| String                              | getString(int parameterIndex) Retrieves the value of a JDBC CHAR, VARCHAR, or LONGVARCHAR parameter as a String in the Java programming language. |
| Time                                | getTime(int parameterIndex) Gets the value of a JDBC TIME parameter as a java.sql.Time object.                                                    |

### Explanation of JDBC Steps:

- **Loading Driver**

**Loading Database driver is very first step towards making JDBC connectivity with the database. It is necessary to load the JDBC drivers before attempting to connect to the database.** The JDBC drivers automatically register themselves with the JDBC system when loaded. Here is the code for loading the JDBC driver:  
Class.forName(driver).newInstance();

- **Establishing Connection**

In the above step we have loaded the database driver to be used. Now its time to make the connection with the database server. In the Establishing Connection step we will logon to the database with user name and password. Following code we have used to make the connection with the database:

```
con = DriverManager.getConnection(url+db, user, pass);
```

- **Executing Statements**

In the previous step we established the connection with the database, now its time to execute query against database. You can run any type of query against database to perform database operations. In this example we will select all the rows from employee table. Here is the code that actually execute the statements against database:

```
ResultSet res = st.executeQuery( "SELECT * FROM employee" );
```

- **Getting Results**

In this step we receives the result of execute statement. In this case we will fetch the employees records from the recordset object and show on the console. Here is the code:

```
while (res.next()) {  
    String employeeName = res.getInt( "employee_name" );  
    System.out.println( employeeName );  
}
```

- **Closing Database Connection**

Finally it is necessary to disconnect from the database and release resources being used. If you don't close the connection then in the production environment your application will fail due to hanging database connections. Here is the code for disconnecting the application from database:

```
con.close();
```

## **Exceptions**

- a). SQLException: It is thrown by the methods whenever there is a problem while accessing the data or any other things.
- b). SQLWarning: This exception is thrown to indicate the warning.
- c). BatchUpdateException: This exception is thrown to indicate that all commands in a batch update are not executed successfully.
- d). DataTruncation: It is thrown to indicate that the data may have been truncated.

## **Categorization of SQL Exceptions**

JDBC 4.0 has introduced two categories of SQLException: SQLTransientException and SQLNonTransientException.

A SQLNonTransientException would be thrown in instances where a retry of the same operation would fail unless the cause of the SQLException is corrected. The following Exceptions have been added, each extending SQLNonTransientException:

- SQLFeatureNotSupportedException
- SQLNonTransientConnectionException
- SQLDataException
- SQLIntegrityConstraintViolationException
- SQLInvalidAuthorizationException
- SQLSyntaxErrorException

A SQLTransientException will be thrown in situations where a previously failed operation might be able to succeed when the operation is retried without any intervention by application-level functionality. The following Exceptions have been added, each extending SQLTransientException:

- SQLTransientConnectionException
- SQLTransactionRollbackException
- SQLTimeoutException

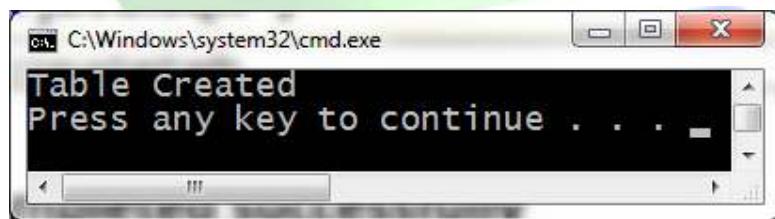
## Examples

**Note:**First Create Database in Ms Access in d: derive name Softech.mdb and then run the following program.

**//Program to Create Table**

**Program:createtab.java**

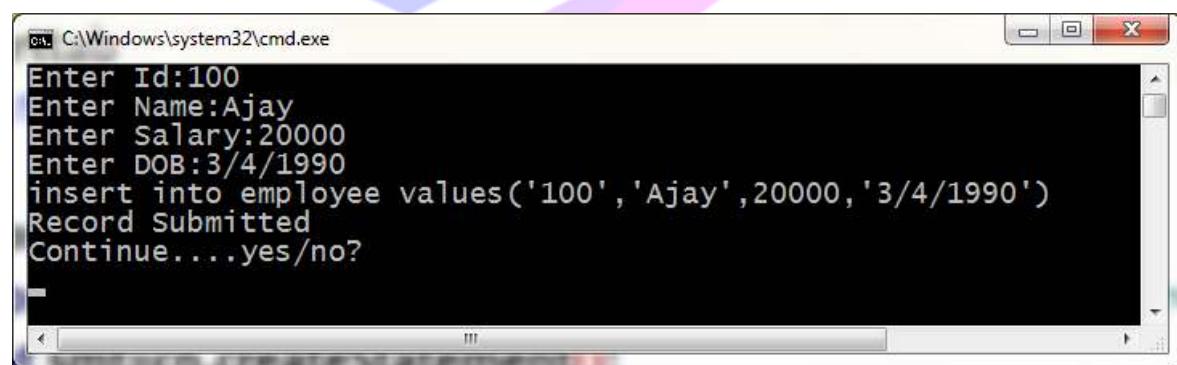
```
import java.sql.*;  
class createtab  
{    public static void main(String arg[])  
    { try{  
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
        Connection cn=DriverManager.getConnection("jdbc:odbc:Driver=Microsoft Access Driver (*.mdb);DBQ=d:/softech.mdb"); //open connection  
        Statement smt=cn.createStatement(); //create statement to supply sql to connection  
        smt.executeUpdate("create table employee(eid text(5) primary key,ename text(20),esal integer,dob date)"); //supply sql to connection  
        System.out.println("Table Created");  
        cn.close();  
    }catch(Exception e)  
    {System.out.println(e);  
    } } }
```



**//Program to Insert record in table**

**Program:inserttab.java**

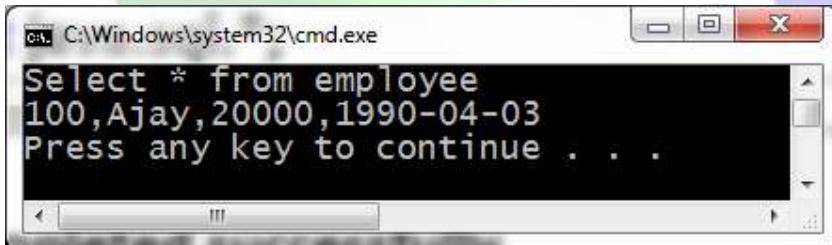
```
import java.sql.*;
import java.io.*;
class inserttab
{
    public static void main(String arg[])
    {
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection cn=DriverManager.getConnection("jdbc:odbc:Driver=Microsoft Access Driver (*.mdb);DBQ=d:/softtech.mdb");
            Statement smt=cn.createStatement();
            DataInputStream X=new DataInputStream( System.in);
            String ch=new String();
            do{
                System.out.print("Enter Id:");
                String id=X.readLine();
                System.out.print("Enter Name:");
                String en=X.readLine();
                System.out.print("Enter Salary:");
                String es=X.readLine();
                System.out.print("Enter DOB:");
                String ed=X.readLine();
                String q="insert into employee values(\""+id+"','"+en+"','"+es+"','"+ed+"\")";
                System.out.println(q);
                smt.executeUpdate(q);
                System.out.println("Record Submitted");
                System.out.println("Continue....yes/no?");
                ch=X.readLine();
            }while(ch.equals("yes"));
            cn.close();
        } catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```



```

//Program to display All Records
//Program:displayrec.java
import java.sql.*;
class displayrec
{
    public static void main(String arg[])
    { try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection
cn=DriverManager.getConnection("jdbc:odbc:Driver=Microsoft Access Driver
(*.mdb);DBQ=d:/softtech.mdb");
        Statement smt=cn.createStatement();
        String q="Select * from employee";
        System.out.println(q);
        ResultSet rs=smt.executeQuery(q);
        if(rs.next())
        {do{
System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3)+" "+rs.getDate(4));
        }while(rs.next());
        }
        else
        {System.out.println("Record Not Found");
        }
        cn.close();
    }
    catch(Exception e)
    {
        System.out.println(e);
    }}}

```

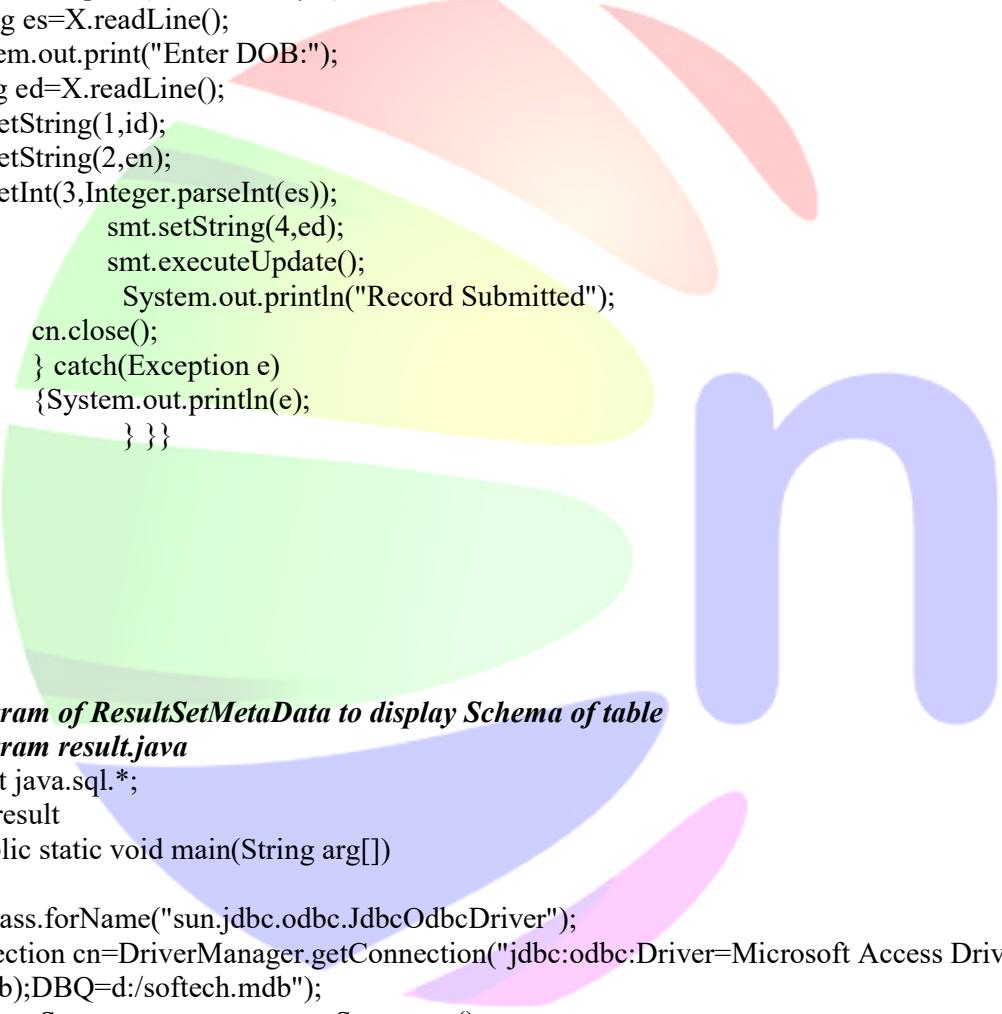


```

//Program to Insert Record Using PreparedStatement
//Program: inserpred.java
import java.sql.*;
import java.io.*;
class inserpred
{
    public static void main(String arg[])
    { try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

```

```
Connection cn=DriverManager.getConnection("jdbc:odbc:Driver=Microsoft Access Driver (*.mdb);DBQ=d:/softtech.mdb");
//parameterized query using PreparedStatement
PreparedStatement smt=cn.prepareStatement("insert into employee
values(?, ?, ?, ?)");
DataInputStream X=new DataInputStream(System.in);
System.out.print("Enter Id:");
String id=X.readLine();
System.out.print("Enter Name:");
String en=X.readLine();
System.out.print("Enter Salary:");
String es=X.readLine();
System.out.print("Enter DOB:");
String ed=X.readLine();
smt.setString(1,id);
smt.setString(2,en);
smt.setInt(3,Integer.parseInt(es));
smt.setString(4,ed);
smt.executeUpdate();
System.out.println("Record Submitted");
cn.close();
} catch(Exception e)
{System.out.println(e);
}}
```



```
//program of ResultSetMetaData to display Schema of table
//program result.java
import java.sql.*;
class result
{
    public static void main(String arg[])
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection cn=DriverManager.getConnection("jdbc:odbc:Driver=Microsoft Access Driver (*.mdb);DBQ=d:/softtech.mdb");
        Statement smt=cn.createStatement();
        String q="Select * from employee";
        ResultSet rs=smt.executeQuery(q);
        if(rs.next())
        {
            ResultSetMetaData rsm=rs.getMetaData();

            for(int i=1;i<=rsm.getColumnCount();i++)
            {System.out.println(rsm.getColumnName(i)+" "+rsm.getColumnTypeName(i)+" "+rsm.getPrecision(i)+" "+rsm.getColumnType(i));}
```

```
    } }
    cn.close();
}
catch(Exception e)
{
    System.out.println(e);
    }}}
```

**output**  
**eid,VARCHAR,5,12**  
**ename,VARCHAR,20,12**  
**esal,INTEGER,10,4**  
**dob,DATETIME,19,93**  
Press any key to continue . . .

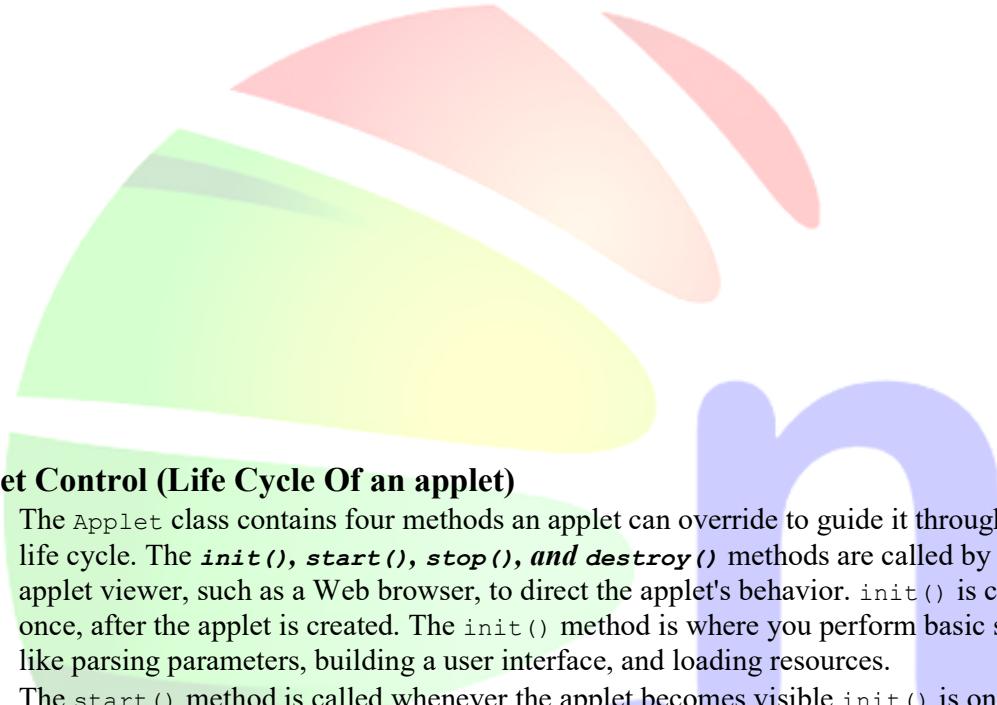
*//program to call database procedure/function prepare in oracle using CallableStatement*  
*//Program:callpro.java*

```
import java.sql.*;
import java.io.*;
class callpro
{ public static void main(String arg[])
{ try
 { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 Connection cn=DriverManager.getConnection("jdbc:odbc:Driver=Microsoft ODBC
for Oracle","internal","oracle");
 CallableStatement smt=cn.prepareCall("{call insemp(?, ?, ?, ?)}");
 DataInputStream X=new DataInputStream(System.in);
 System.out.print("Enter Id:");
 String id=X.readLine();
 System.out.print("Enter Name:"); String en=X.readLine();
 System.out.print("Enter Salary:"); String es=X.readLine();
 System.out.print("Enter DOB:"); String ed=X.readLine();
 smt.setString(1,id);
 smt.setString(2,en);
 smt.setInt(3,Integer.parseInt(es));
 smt.setString(4,ed);}}
```

```
        smt.execute();
        cn.close(); }catch(Exception e)
{System.out.println(e); }}
```



# Applet



## Applet Control (Life Cycle Of an applet)

- The `Applet` class contains four methods an applet can override to guide it through its life cycle. The `init()`, `start()`, `stop()`, and `destroy()` methods are called by an applet viewer, such as a Web browser, to direct the applet's behavior. `init()` is called once, after the applet is created. The `init()` method is where you perform basic setup like parsing parameters, building a user interface, and loading resources.
- The `start()` method is called whenever the applet becomes visible `init()` is only called once in the life of an applet, but `start()` and `stop()` can be called any number of times (but always in the logical sequence). For example, `start()` is called when the applet is displayed, such as when it scrolls onto the screen; `stop()` is called if the applet scrolls off the screen or the viewer leaves the document. `start()` tells the applet it should be active.
- The `public void paint(Graphics g)` method invokes implicitly when applets start to paint some graphics on applet window(Canvas).`Graphics` class contains various methods to draw line,image and text on canvas with color & font effects.To call the `paint()` method again one can use `repaint()` method,but this will clears the previous contents.
- There is another method `public void update(Graphics g)` method which invokes explicitly through `repaintmethod` to paint some graphics on applet window(Canvas). To call the `upadte()` method again one can use `repaint()` method,but this will not clears the previous contents.
- The `destroy()` method is called to clean up resources of applets after stop method.

Example

```
import java.applet.*;  
import java.awt.*;
```

```
public class first extends Applet
{
    public void init()
    {
    }
    public void start()
    {
    }

    public void paint(Graphics g)
    {
    }
    public void update(Graphics g)
    {
    }
    public void stop()
    {
    }
    public void destroy()
    {
    }
}
```

## The Applet security Sandbox

Applets are controlled within the browser by an applet `SecurityManager`. The `SecurityManager` is part of the application that runs the applet, e.g., the web browser or applet viewer. It is installed before the browser loads any applets and implements the basic restrictions that let you run untrusted applets safely. Remember that, aside from basic language robustness, there are no inherent security restrictions on a standalone Java application. It is the browser's responsibility to install a special security manager and limit what applets are allowed to do.

Most browsers impose the following restrictions on untrusted applets:

- Untrusted Applets cannot read or write files on the local host.
- Untrusted Applets can only open network connections (sockets) to the server from which they originated.
- Untrusted Applets cannot start other processes on the local host.
- Untrusted Applets cannot have native methods.



# AWT

# Containers

# Component

## **GUI**

A class library is provided by the Java programming language which is known as **Abstract Window Toolkit (AWT)**. The Abstract Window Toolkit (AWT) contains several **graphical class and interfaces** which can be added and positioned to the display area with a layout manager.

As the Java programming language, the AWT is also platform-independent. A common set of tools is provided by the AWT for graphical user interface design. .

## **AWT Basics**

The superclass of all graphical user interface objects is **Component**. The components are added to the Container object to create a Graphical User Interface (GUI). AWT provides various user interface component like textbox,label,radiobutton,checkbox,button etc. to prepare interactive user interface.

## **Basic GUI Logic**

The GUI application or applet is created in three steps. These are:

- **Add components to Container objects to make your GUI.**
- **Then you need to setup event handlers for the user interaction with GUI.**
- **Explicitly display the GUI for application.**

## **What is AWT in java**

In this section, you will learn about the `java.awt.*`; package available with JDK. AWT stands for Abstract Windowing Toolkit. It contains all classes to write the program that interface between the user and different windowing toolkits. You can use the AWT package to develop user interface objects like buttons, checkboxes, radio buttons and menus etc.

Following some components of Java AWT are explained :

1. **Labels** : This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in your application and label never perform any type of action. Syntax for defining the label only and with justification :

```
Label label_name = new Label ("This is the label text.");
```

Above code simply represents the text for the label.

```
Label label_name = new Label ("This is the label text.", Label.CENTER);
```

Justification of label can be left, right or centered. Above declaration used the center justification of the label using the Label.CENTER.

2. **Buttons** : This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for your application. The syntax of defining the button is as follows :

```
Button button_name = new Button ("This is the label of the button.");
```

You can change the Button's label or get the label's text by using the

**Button.setLabel(String)** and **Button.getLabel()** method. Buttons are added to the it's container using the add (button\_name) method.

3. **Check Boxes** : This component of Java AWT allows you to create check boxes in your applications. The syntax of the definition of Checkbox is as follows :

```
CheckBox checkbox_name = new CheckBox ("Optional check box 1", false);
```

Above code constructs the unchecked Checkbox by passing the boolean valued argument *false* with the Checkbox label through the Checkbox() constructor. Defined Checkbox is added to it's container using add (checkbox\_name) method. You can change and get the checkbox's label using the setLabel (String) and getLabel() method. You can also set and get the state of the checkbox using the setState(boolean) and getState() method provided by the **Checkbox** class.

4. **Radio Button** : This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes which group name is same. Only one Checkbox from a Checkbox Group can be selected at a time. Syntax for creating radio buttons is as follows :

```
CheckboxGroup chkgp = new CheckboxGroup();  
add (new Checkbox ("One", chkgp, false);  
add (new Checkbox ("Two", chkgp, false);  
add (new Checkbox ("Three", chkgp, false);
```

In the above code we are making three check boxes with the label "One", "Two" and "Three". If you mention more than one true valued for checkboxes then your program takes the last true and show the last check box as checked.

5. **Text Area:** This is the text container component of Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

```
TextArea txtArea_name = new TextArea();
```

You can make the Text Area editable or not using the setEditable (boolean) method. If you pass the boolean valued argument *false* then the text area will be non-editable otherwise it will be editable. The text area is by default in editable mode. Text are set in the text area using the setText(string) method of the **TextArea** class.

6. **Text Field:** This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows :

```
TextField txtfield = new TextField(20);
```

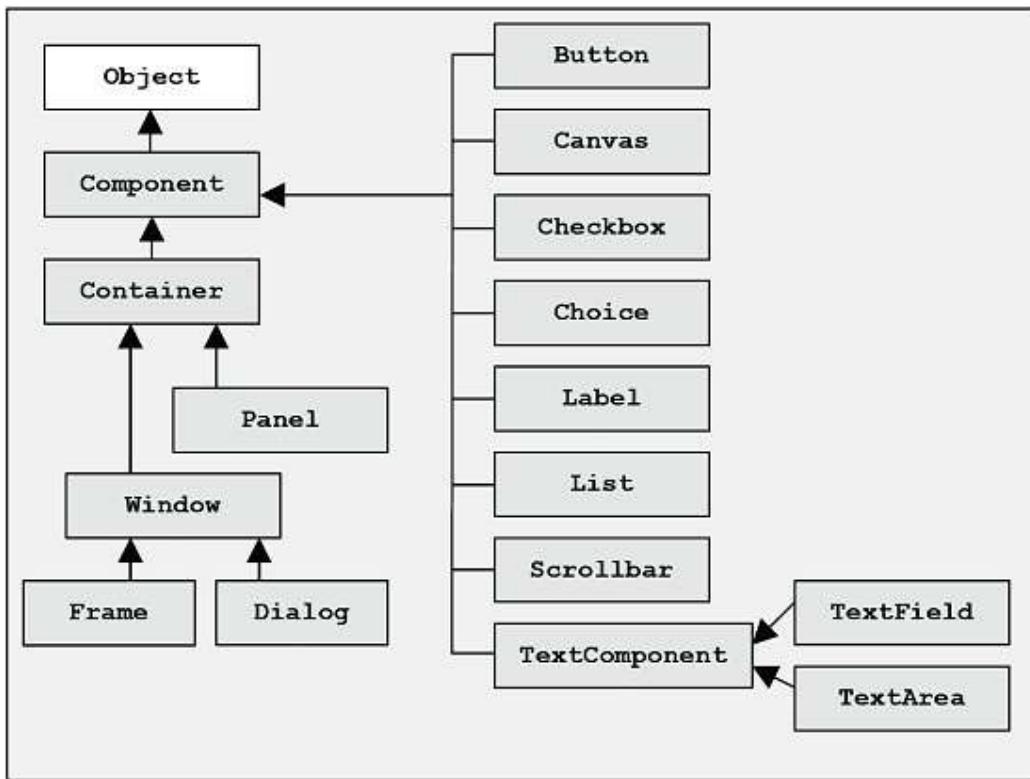
You can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

### **Component:**

*A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.*

The `Component` class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components. Class `Component` can also be extended directly to create a lightweight component.

The following diagram shows most of the the class hierarchy of the AWT (Abstract Windows Toolkit), which comes as part of the core Java language (`java.awt` package and sub-packages.)



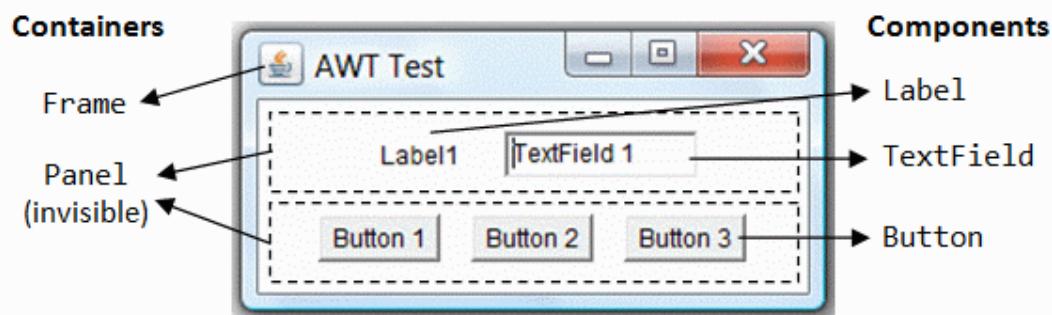
### Containers

The `Container` class provides for holding instances of other component classes. The `Window` subclass, for example, provides for the *top level* visible containers, **Frame and Dialog**, **Window and Panels** that hold various visible components.

Containers can hold other containers. The `Panel` class, in particular, is used within a top level container to arrange its sub-components, which often are also panels. An elaborate GUI display with lots of buttons, textfields, and other components will employ several panels and sub-panels to arrange the visible *atomic* (non-container) components.

As seen in the diagram above, the basic AWT includes several atomic components such as buttons, labels, and textfields. You can create fairly elaborate GUI displays with these tools.

### Container vs. Component

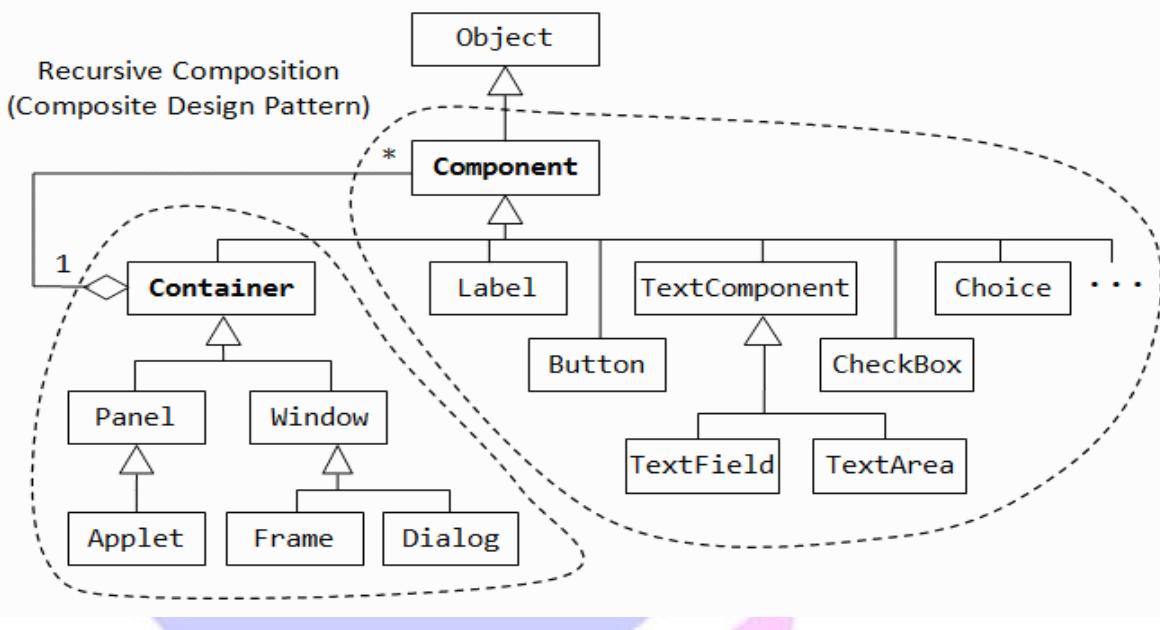


Two groups of classes are involved in a GUI program:

1. *Component classes*: Components are elementary UI entities such as buttons, labels, text fields, and check boxes.
2. *Container classes*: Containers, such as frames and panels, are used to hold UI components. A container can also hold containers.

In the above example, there are three containers: a `Frame` and two `Panel`s. `Frame` is the *top-level container* of an AWT GUI program. `Frame` has a title bar (containing program icon, title, minimize, maximize/restore-down and close buttons), an optional menu bar and program display area. `Panel` is an *invisible rectangular box* used to group related UI component. In the above example, the top-level `Frame` contains two invisible `Panel`s. There are five components: a `Label` (which provides text description), a `TextField` (for users to enter text), and three `Button`s (for activating certain programmed actions).

## AWT Hierarchy of Classes



As mentioned earlier, there are two groups of classes in the AWT hierarchy: containers and components. A container (e.g., `Frame` and `Panel`) holds components (e.g., `Button`, `TextField`). A container (e.g., `Frame` and `Panel`) can also hold other containers (e.g. `Panel`). Hence, we have a situation that "a container can contain containers or components".

This is quite a common problem: e.g., a directory contains (sub)directories or files; a group contains (sub)groups or elementary elements; and the tree data structure. A *design pattern* has been proposed for this problem. A design pattern is basically a proven and possibly the best solution for a specific type of common problem.

As shown in the class diagram, there are two sets of relationship between `Container` and `Component` classes.

1. *One-to-many aggregation*: A `Container` contains zero or more `Components`. Each `Component` is contained in exactly one `Container`.

2. *Generalization (or Inheritance):* Container is a *subclass* of Component. In other words, a Container is a Component, which possesses all the properties of Component and can be substituted in place of a Component.

Combining both relationships, we have: A Container contains Components. Since a Container is a Component, a Container can also contain Containers. Consequently, a Container can contain Containers and Components.



# AWT Control With Examples



## Applet Class

The Applet class can be used to create web page applets.

### Package

Java.applet  
Common Public Constructors  
**Applet ()**

Creates a blank Applet instance.

## **Common Public Methods**

### **Container getContentPane ()**

Returns the contentPane object for this applet.

### **int getHeight ()**

Returns the vertical size of the applet.

### **int getWidth ()**

Returns the horizontal size of the applet.

### **void init ()**

The *init* method is called once after the constructor is executed.

### **void paint (Graphics g)**

The *paint* method is called every time the the applet is to be redrawn.

### **void start ()**

The *start* method is called when the web page or appletviewer is redisplayed.

### **void setBackground (Color BgColor)**

Sets the background color of the text for the applet.

## **Arguments**

### **BgColor**

The color to be used for the background of the applet.

## **Source Code - Applet1.java**

```
import java.awt.*;
import java.applet.*;

public class Applet1 extends Applet {

    public void paint (Graphics g) {
        setBackground (Color.cyan);
        g.setColor (Color.red);

        g.drawLine (200, 200, 300, 200);

    } }

/*
<applet code=Applet1 width=300 height=300>
</applet>
*/
```

## **Graphics Class**

### **Usage:**

The Graphics class is the basic drawing canvas for Java programs.

### **Package:**

**Java.awt**

## **Common Constructors:**

`Graphics` is an abstract class, and cannot be constructed. A `Graphics` object is passed to the Applet `paint` method as an argument.

## **Common Methods:**

```
clearRect (int X, int Y, int Width, int Height)
clipRect (int X, int Y, int Width, int Height)
drawArc (int X, int Y, int Width, int Height, int Theta, int Delta)
drawImage (Image Img, int X, int Y, ImageObserver Observer)

drawImage (Image Img, int X, int Y, int Width, int Height,
ImageObserver Observer)
    drawLine (int X1, int Y1, int X2, int Y2)
drawOval (int X, int Y, int Width, int Height)
drawPolygon (int[] XArray, int[] YArray, int N)
drawPolyline (int[] XArray, int[] YArray, int N)
drawRect (int X, int Y, int Width, int Height)
drawRoundRect (int X, int Y, int Width, int Height, int OvalWidth,
int OvalHeight)
drawString (java.lang.String Text, int X, int Y)
fill3DRect (int X, int Y, int Width, int Height, boolean Raised)
fillArc (int X, int Y, int Width, int Height, int Theta, int Delta)
fillOval (int X, int Y, int Width, int Height)
fillPolygon (int[] XArray, int[] YArray, int N)
fillRect (int X, int Y, int Width, int Height)
fillRoundRect (int X, int Y, int Width, int Height, int OvalWidth,
int OvalHeight)
setColor (Color C)
setFont (Font F)
```

## **Drawing Arcs**

### **Usage:**

Vertical location of the upper left corner of the rectangle, in pixels. The `Graphics` `drawArc` method can be used to draw arcs which are portions of ellipses, and the `Graphics` `fillArc` method can be used to draw pie shapes which are portions of filled ellipses.

### **Class:**

`Graphics`

### **Example Methods:**

```
drawArc (int X, int Y, int Width, int Height, int Theta, int Delta)
fillArc (int X, int Y, int Width, int Height, int Theta, int Delta)
```

### **Arguments:**

`Delta`

The size of the arc, in rectangular degrees.

`Height`

The vertical size of the oval, in pixels.

`Theta`

The beginning angle of the arc, in rectangular degrees.

Width

The horizontal size of the oval, in pixels.

X

Horizontal location of the upper left corner of the bounding rectangle, in pixels.

Y

Vertical location of the upper left corner of the bounding rectangle, in pixels.

### Example:

#### Code:

```
import java.applet.Applet;
import java.awt.*;

public class Arc1 extends Applet {

    public void paint (Graphics g) {
        g.drawArc (20, 30, 150, 130, 45, 90);
        g.fillArc (20, 30, 150, 130, 240, 60);
    }
}
```

## Drawing Images

#### Usage:

An image (.jpg, .jpeg, or .gif) can be displayed using the drawImage method after being loaded with the getImage method of the Applet.

#### Class:

Graphics

#### Example Methods:

```
drawImage (Image Img, int X, int Y, ImageObserver Observer)
drawImage (Image Img, int X, int Y, int Width, int Height,
ImageObserver Observer)
```

#### Arguments:

Height

The vertical size of the bounding rectangle in pixels.

Width

The horizontal size of the bounding rectangle in pixels.

X

Horizontal location of the upper left corner of the bounding rectangle, in pixels.

Y

Vertical location of the upper left corner of the bounding rectangle, in pixels.

### Example:

#### Code:

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;

public class Image1 extends Applet {
    private Image Picture;
```

```

public void init () {
    URL Base;
    Base = getDocumentBase ();
    Picture = getImage (Base, ".../.../images/stache.gif");
}

public void paint (Graphics g) {
    g.drawImage (Picture, 10, 20, this);
}

}

```

## Drawing Ovals

### Usage:

The Graphics drawOval method can be used to draw ovals, and Graphics fillOval method can be used to draw filled ovals. In both cases, the location of the oval is defined by the upper left corner of a rectangle which bounds the oval.

### Class:

Graphics

### Example Methods:

```

drawOval (int X, int Y, int Width, int Height)
fillOval (int X, int Y, int Width, int Height)

```

### Arguments:

Height

The vertical size of the oval, in pixels.

Width

The horizontal size of the oval, in pixels.

X

Horizontal location of the upper left corner of the bounding rectangle, in pixels.

Y

Vertical location of the upper left corner of the bounding rectangle, in pixels.

### Example:

### Code:

```

import java.applet.Applet;
import java.awt.*;

public class Oval1 extends Applet {

    public void paint (Graphics g) {
        g.drawOval (20, 30, 130, 50);
        g.fillOval (20, 100, 130, 50);
    }
}

```

## Drawing Polygons

### Usage

- The Graphics drawPolygon method can be used to draw outline polygons, and the Graphics fillPolygon method can be used to draw filled polygons.

## Class

`java.awt.Graphics`

## Prototype

```
void drawPolygon (int[] XArray, int[] YArray, int N)
void fillPolygon (int[] XArray, int[] YArray, int N)
```

## Arguments

### XArray

Array of the horizontal locations of the vertices of the polygon, in pixels.

### YArray

Array of the vertical locations of the vertices of the polygon, in pixels.

### N

Number of points.

## Applet - Polygon1

### Source Code - Polygon1.java

```
import java.applet.Applet;
import java.awt.*;

public class Polygon1 extends Applet {

    public void paint (Graphics g) {
        int Cursor;
        int[] XArray = {20, 160, 120, 160, 20, 60};
        int[] YArray = {20, 20, 90, 160, 160, 90};
        g.drawPolygon (XArray, YArray, 6);

    }
}
```

## Drawing Rectangles

The Graphics drawRect method can be used to draw outline rectangles, and the Graphics fillRect method can be used to draw filled rectangles.

## Class

`java.awt.Graphics`

## Example Methods

```
void drawRect (int X, int Y, int Width, int Height)  
void fillRect (int X, int Y, int Width, int Height)
```

## Arguments

`Height`

The vertical size of the rectangle in pixels.

`Width`

The horizontal size of the rectangle in pixels.

`X`

Horizontal location of the upper left corner of the rectangle, in pixels.

`Y`

Vertical location of the upper left corner of the rectangle, in pixels.

## Example

### Code

```
import java.applet.Applet;  
import java.awt.*;  
  
public class Rect1 extends Applet {  
  
    public void paint (Graphics g) {  
        g.drawRect (20, 30, 130, 50);  
        g.fillRect (20, 100, 130, 50);  
    }  
}
```

## Drawing Rounded Rectangles

### **Usage:**

The Graphics drawRoundRect method can be used to draw outline rectangles with rounded corners, and the Graphics fillRoundRect method can be used to draw filled rectangles with rounded corners.

### **Class:**

Graphics

### **Example Methods:**

```
drawRoundRect (int X, int Y, int Width, int Height, int OvalWidth,  
int OvalHeight)  
fillRoundRect (int X, int Y, int Width, int Height, int OvalWidth,  
int OvalHeight)
```

### **Arguments:**

Height

The vertical size of the rectangle in pixels.

OvalHeight

The vertical size of the oval from which the corners are formed, in pixels.

OvalWidth

The horizontal size of the oval from which the corners are formed, in pixels.

Width

The horizontal size of the rectangle in pixels.

X

Horizontal location of the upper left corner of the rectangle, in pixels.

Y

Vertical location of the upper left corner of the rectangle, in pixels.

### **Example:**

```
import java.applet.Applet;  
import java.awt.*;  
  
public class RoundRect1 extends Applet {  
  
    public void paint (Graphics g) {  
        g.drawRoundRect (20, 30, 130, 50, 40, 40);  
        g.fillRoundRect (20, 100, 130, 50, 60, 30);  
    }  
}
```

## **Drawing 3DRectangles**

### **Usage:**

The Graphics fill3DRect method can be used to draw filled rectangles, which is shaded to appear either raised or lowered.

### **Class:**

Graphics

## **Example Methods:**

```
fill3DRect (int X, int Y, int Width, int Height, boolean Raised)
```

### **Arguments:**

Height

The vertical size of the rectangle in pixels.

Raised

If true, the "button" appears raised; if false, the "button" appears lowered.

Width

The horizontal size of the rectangle in pixels.

X

Horizontal location of the upper left corner of the rectangle, in pixels.

Y

Vertical location of the upper left corner of the rectangle, in pixels.

```
import java.applet.Applet;
import java.awt.*;

public class Rect3D1 extends Applet {

    public void paint (Graphics g) {
        g.setColor (Color.green);
        g.fill3DRect (20, 20, 130, 50, true);
        g.fill3DRect (170, 20, 130, 50, false);
    }
}
```

## **Drawing Strings**

The Graphics drawString method can be used to print text.

### **Class**

Graphics

## **Example Methods**

```
drawString (java.lang.String Text, int X, int Y)
```

### **Arguments**

Text

The text do be drawn

X

Horizontal location of the left end of the text baseline, in pixels.

Y

Vertical location of the left end of the text baseline, in pixels.

```
import java.applet.Applet;
import java.awt.*;

public class String1 extends Applet {

    public void paint (Graphics g) {
        g.drawString ("Hello, World", 20, 50);
    }

}
```

## Java Color Example

### Usage:

The Color class can be used to build custom colors for use in graphics.

### Package:

`Java.awt`

### Subclass of:

`Java.lang.Object`

### Common Constructors:

`Color (int Red, int Green, int Blue)`

Creates a color with the designated combination of red, green, and blue (each 0-255).

`Color (float Red, float Green, float Blue)`

Creates a color with the designated combination of red, green, and blue (each 0-1).

`Color (int rgb)`

Creates a color with the designated combination of red, green, and blue (each 0-255).

### Common Methods:

`getBlue ()` Returns the blue component of the color.

`getGreen ()` Returns the green component of the color.

`getRed ()` Returns the red component of the color.

### Arguments:

`Blue`

The blue component of the color. As an integer, the range is 0-255; as a float, the range is 0-1.

`Green`

The green component of the color. As an integer, the range is 0-255; as a float, the range is 0-1.

`Red`

The red component of the color. As an integer, the range is 0-255; as a float, the range is 0-1.

`rgb`

The red, green, and blue components of a color, packed into a single 32 bit integer, with red as bits 23-16, green as bits 15-8, and blue as bits 7-0;

### Example Code:

```
import java.awt.*;
import java.applet.*;
```

```
public class Color1 extends Applet {  
  
    public void paint (Graphics g) {  
        Color c = new Color (Red, Green, Blue);  
        g.setColor (c);  
        g.drawLine (100,100,200,200);  
    }  
}  
}
```

## Setting the Drawing Color

### Usage:

The drawing color can be set with the Graphics setColor method.

### Class:

Graphics

### Example Methods:

**setColor (Color C)**

### Arguments:

C

The new rendering color.

### Example Code:

```
import java.applet.Applet;  
import java.awt.*;  
  
public class Color1 extends Applet {  
  
    public void paint (Graphics g) {  
        g.fillRect (10, 10, 30, 80);  
        g.setColor (Color.red);  
        g.fillRect (40, 10, 30, 80);  
        g.setColor (Color.green);  
        g.fillRect (70, 10, 30, 80);  
        g.setColor (Color.blue);  
        g.fillRect (100, 10, 30, 80);  
        g.setColor (Color.white);  
        g.fillRect (130, 10, 30, 80);  
    }  
}
```

## Font Class

### Common Constructor:

```
Font (String Name, int Style, int Size)
```

- Where
  - Name is the logical or physical name of the font. Available logical names are:
    - "Dialog" - Default type for dialog boxes
    - "DialogInput" - Default type for dialog box inputs
    - "Monospaced" - Type with all characters having the same width
    - "Serif" - Type with serifs
    - "SansSerif" - Type without serifs
  - Style is the type face:
    - Font.PLAIN - plain face
    - Font.BOLD - **bold face**
    - Font.ITALIC - *italic face*
    - Font.BOLD | Font.ITALIC - **bold italic face**
  - Size is the *nominal* size of the font in points.

### Common Methods:

```
String getFamily ()
```

Returns the family name of this Font.

```
String getFontName ()
```

Returns the font face name of this Font.

```
String getName ()
```

Returns the logical name of this Font.

```
int getSize ()
```

Returns the point size of this Font, rounded to an integer.

```
int getStyle ()
```

Returns the style of this Font.

## Setting Text Fonts

### Usage:

The Graphics drawLine method can be used to print text.

### Class:

Graphics

### Example Methods:

```
setFont (Font F)
```

### Arguments:

F

A text font to be used for future text drawing.

### Example:

### Code:

```
import java.applet.Applet;
```

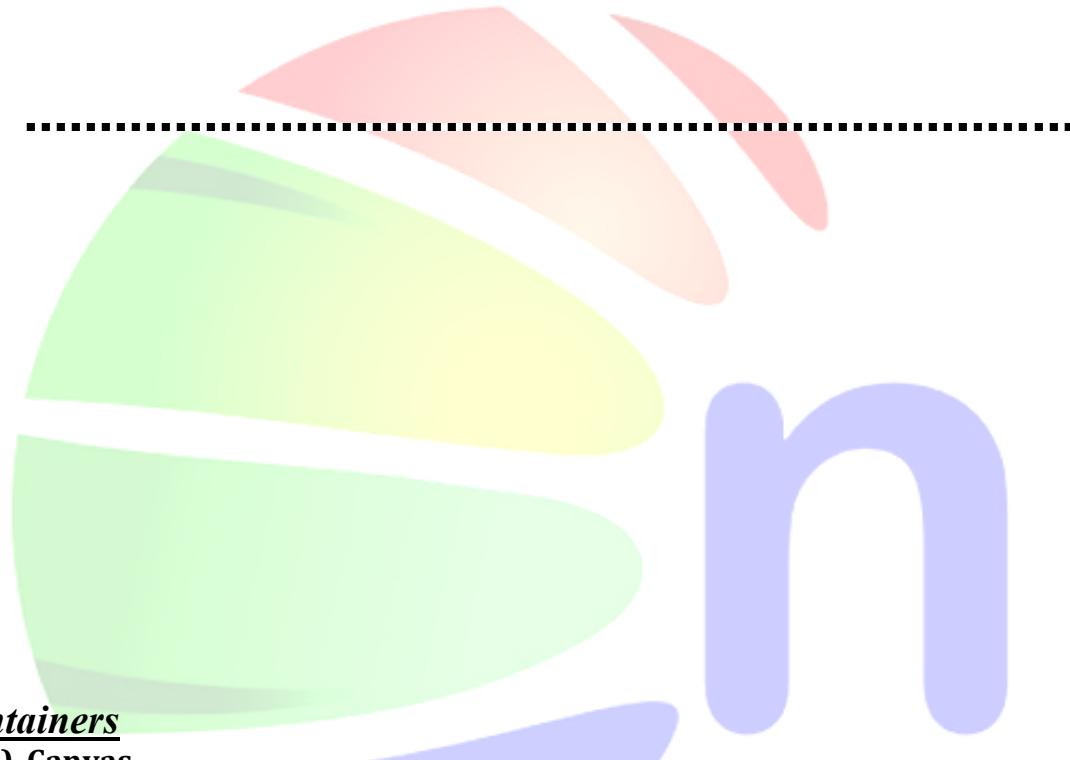
```

import java.awt.*;

public class Font1 extends Applet {

    public void paint (Graphics g) {
        Font CurrentFont;
        g.drawString ("Hello, World", 20, 20);
        CurrentFont = new Font ("serif", Font.PLAIN, 24);
        g.setFont (CurrentFont);
        g.drawString ("Hello, World", 20, 50);
        CurrentFont = new Font ("monospaced", Font.ITALIC, 18);
        g.setFont (CurrentFont);
        g.drawString ("Hello, World", 20, 80);
    }
}

```



## Containers

### 1) Canvas

A `Canvas` is a class just waiting to be subclassed. Through `Canvas`, you can create additional AWT objects that are not provided by the base classes. `Canvas` is also useful as a drawing area, particularly when additional components are on the screen. It is tempting to draw directly onto a `Container`, but this often isn't a good idea. Anything you draw might disappear underneath the components you add to the container. When you are drawing on a container, you are essentially drawing on the background. The container's layout manager doesn't know anything about what you have drawn and won't arrange components with your artwork in mind. To be safe, do your drawing onto a `Canvas` and place that `Canvas` in a `Container`.

### 2) Panel

to make Applet or Frame layout easier, you break a frame up into regions and compose each of them separately. Each region is called a `Panel`. Each can have its own different `LayoutManager`.`Panels` don't have any visible bounding lines. You can delimit them with differing background colours. If you want something to draw on with `drawString` and `drawLine` normally you would use a `Canvas`.

**Panel** is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.  
The default layout manager for a panel is the `FlowLayout` layout manager.

## Constructors

```
public Panel()
```

Creates a new panel using the default layout manager. The default layout manager for all panels is the `FlowLayout` class.

```
public Panel(LayoutManager layout)
```

Creates a new panel with the specified layout manager.

- **Parameters**
  - `layout` - the layout manager for this panel.

### Methods

```
public void add(Component comp,  
Object constraints)
```

```
public void remove(Component comp)
```

Removes the specified component from this container. This method also notifies the layout manager to remove the component from this container's layout via the `removeLayoutComponent` method.

```
public void removeAll()
```

Removes all the components from this container. This method also notifies the layout manager to remove the components from this container's layout via the `removeLayoutComponent` method.

```
import java.awt.*;  
import java.awt.event.*;  
  
public class panel{  
    public static void main(String[] args){  
        Panel panel = new Panel();  
        panel.add(new Button("Button 1"));  
        panel.add(new Button("Button 2"));  
        panel.add(new Button("Button 3"));  
        Frame frame = new Frame("Container Frame");  
        TextArea txtArea = new TextArea();  
        frame.add(txtArea, BorderLayout.CENTER);  
        frame.add(panel, BorderLayout.SOUTH);  
        frame.setSize(400,400);  
        frame.setVisible(true);  
  
    }  
}
```

## Frame

A Frame is a top-level window with a title and a border. It is normally used as the main window of a standalone application.

### Package

java.awt

### Class

java.awt.Frame

### Common Public Constructors

#### **Frame ()**

Creates a blank Frame instance.

#### **Frame (String Text)**

Creates a Frame instance with the specified text in the title bar.

### Common Public Methods

#### **addWindowListener(WindowListener Handler)**

Configures a window event handler for the frame.

#### **setBackground (Color BackgroundColor)**

Sets the background color of the frame.

#### **setFont (Font TextFont)**

Sets the font for this component.

#### **setForeground (Color TextColor)**

Sets the color of the text for the frame.

#### **setSize (int Width, int Height)**

Resizes this window so that it has the specified Width and Height.

#### **setTitle (String Text)**

Sets the text for the title bar.

#### **show ()**

Makes the window visible.

### Arguments

#### **BackgroundColor**

The color to be used for the background of the button.

#### **Handler**

The object which handles window events from this frame.

### **Text**

The text to appear in the title bar.

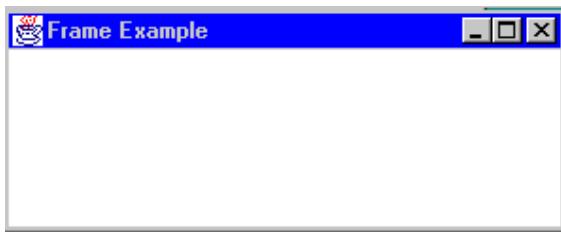
### **TextColor**

The color to be used for the text of the button.

### **TextFont**

The font to be used for the text of the button.

## **Example - Frame2.class**



### **Source Code - Frame2.java**

```
import java.awt.*;
import java.awt.event.*;

public class Frame2 extends Frame {
    Frame2 () {
        setTitle ("Frame Example");
        setSize (300,120);

        show ();
    }
    public static void main (String args[]) {
        Frame f;
        f = new Frame2 ();
    }
}
```

## **Button**

### **Usage**

The Button class can be used to add interactive buttons to a container.

## Package

Java.awt

## Common Public Constructors

### Button ()

Creates a blank Button instance.

### Button (String Text)

Creates a Button instance with the specified text.

## Common Public Methods

### void addActionListener

(ActionListener Handler)

### void setActionCommand

(String ActionText)

### void setBackground (Color BackgroundColor)

Sets the background color of the button.

### void setEnabled (boolean State)

Enables or disables the button.

### void setFont (Font TextFont)

Sets the font for this component.

### void setForeground (Color TextColor)

Sets the color of the text for the button.

### void setLabel (String Text)

Sets the text for the button.

## Arguments

### ActionText

The ActionCommand text for the event associated with the button.

### BackgroundColor

The color to be used for the background of the button.

### Handler

The object which handles action events from this combo box.

### State

The enable state of the button (true or false.)

### Text

The text to appear next to the button.

### TextColor

The color to be used for the text of the button.

### TextFont

The font to be used for the text of the button.

```
//Example Of Button,TextField,Label
```

```
//Program:app2.java
```

```
import java.awt.*;
```

```

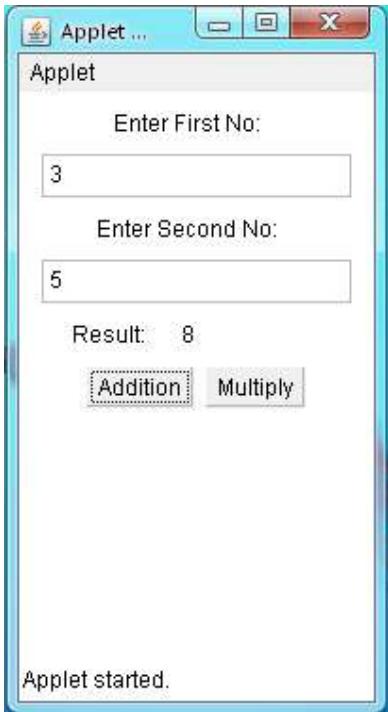
import java.applet.*;
import java.awt.event.*;
public class app2 extends Applet implements ActionListener
{ Label L1=new Label("Enter First No:");
Label L2=new Label("Enter Second No:");
Label L3=new Label("Result:");
Label L4=new Label("See Here....");
TextField T1=new TextField(20);
TextField T2=new TextField(20);
Button B1=new Button("Addition");
Button B2=new Button("Multiply");

public void start()
{
    add(L1);add(T1);
    add(L2);add(T2);
    add(L3);add(L4);
    add(B1);add(B2);
B1.addActionListener(this);
B2.addActionListener(this);
}

public void actionPerformed(ActionEvent e)
{
    int f=Integer.parseInt(T1.getText());
    int s=Integer.parseInt(T2.getText());
    if(e.getSource()==B1)
    {
        int r=f+s;
        L4.setText(r+"");
    }
    else if(e.getSource()==B2)
    {
        int c=f*s;
        L4.setText(c+"");
    }
}

```





## Java AWT CheckBox Examples

### Usage:

The CheckBox class can be used to add interactive grouped radio checkboxes to a container.

### Package:

`java.awt`

### Subclass of:

`Java.awt.Component`

### Common Constructors:

`CheckBox ()`

Creates a blank CheckBox instance.

`CheckBox (String Text)`

Creates a CheckBox instance with the specified text.

`CheckBox (String Text, boolean Selected)`

Creates a CheckBox instance with the specified text and the designated selection state.

`CheckBox (String Text, CheckboxGroup Group, boolean Selected)`

Creates a CheckBox instance with the specified text and the designated selection state as part of a CheckbosGroup. This causes it to interact with other members of the group, so that only one is checked at a time, and the checkbox shape is changed to round.

### Common Methods:

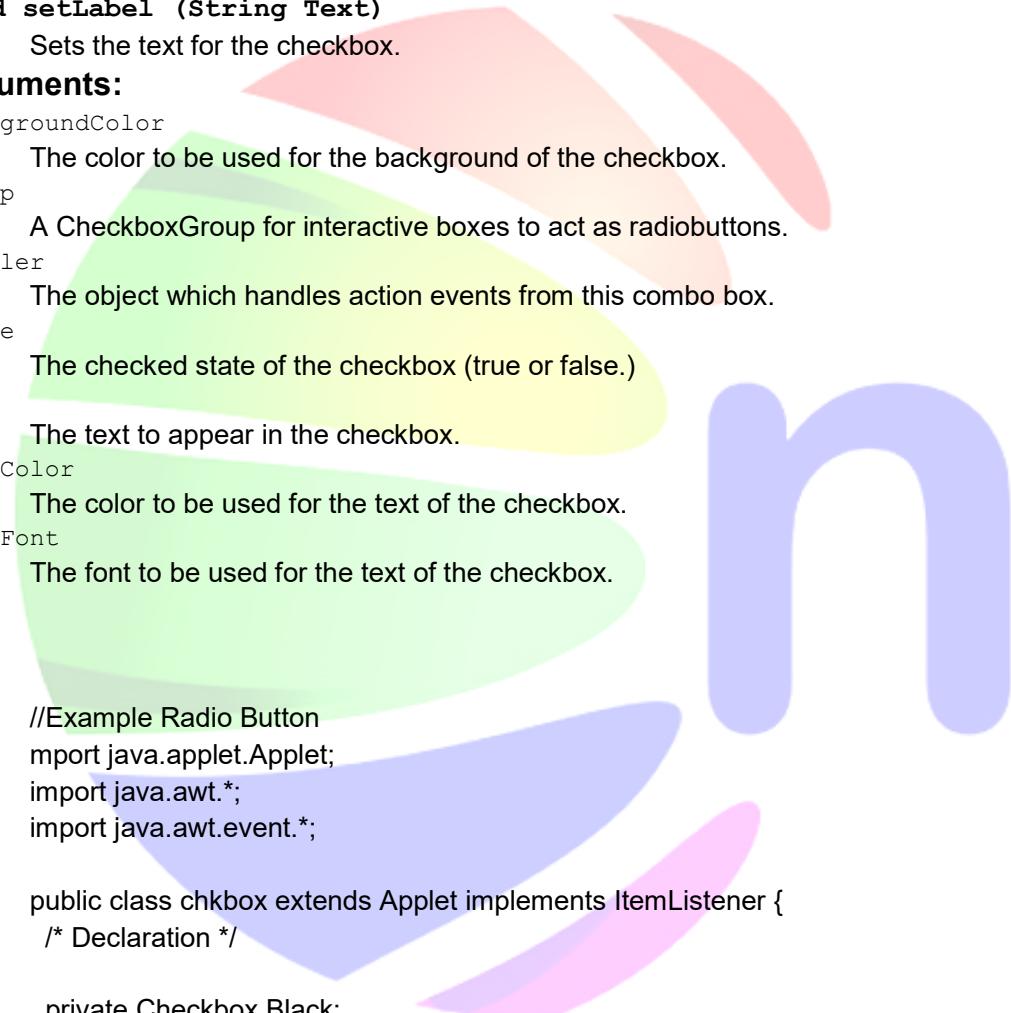
`void addItemListener (ItemListener Handler)`

Configures an event handler for the checkbox.

```
boolean getState()
    Returns the state true or false (checked or not checked) of the checkbox.
void setBackground (Color BackgroundColor)
    Sets the background color of the checkbox.
void setSelected (boolean State)
    Sets the state (checked or not checked) of the checkbox.
void setFont (Font TextFont)
    Sets the font for this component.
void setForeground (Color TextColor)
    Sets the color of the text for the checkbox.
void setLabel (String Text)
    Sets the text for the checkbox.
```

### **Arguments:**

BackgroundColor  
The color to be used for the background of the checkbox.  
Group  
A CheckboxGroup for interactive boxes to act as radiobuttons.  
Handler  
The object which handles action events from this combo box.  
State  
The checked state of the checkbox (true or false.)  
Text  
The text to appear in the checkbox.  
TextColor  
The color to be used for the text of the checkbox.  
TextFont  
The font to be used for the text of the checkbox.



```
//Example Radio Button
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class chkbox extends Applet implements ItemListener {
    /* Declaration */

    private Checkbox Black;
    private Checkbox Red;
    private Checkbox Blue;
    private CheckboxGroup TextColor;
    private TextField Result;

    public chkbox () {
        /* Instantiation */

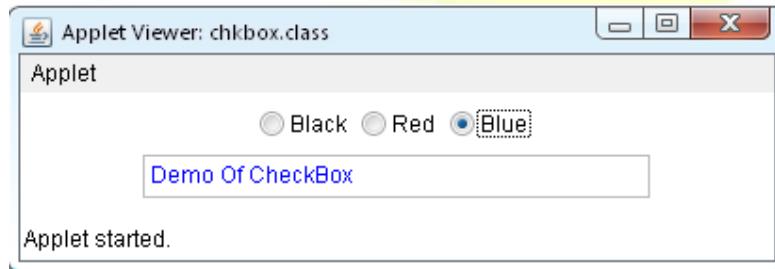
        TextColor = new CheckboxGroup();
```

```

Black = new Checkbox ("Black", TextColor, true);
Blue = new Checkbox ("Blue", TextColor, false);
Red = new Checkbox ("Red", TextColor, false);
Result = new TextField ("Demo Of CheckBox", 35);
    add (Black);
    add (Red);
    add (Blue);
    add (Result);

/* Configuration */
Black.addItemListener (this);
Red.addItemListener (this);
Blue.addItemListener (this);
}
public void itemStateChanged(ItemEvent e) {
    if (Black.getState()) {
        Result.setForeground (Color.black);
    } else if (Red.getState()) {
        Result.setForeground (Color.red);
    } else if ( Blue.getState()) {
        Result.setForeground (Color.blue);
    }  }}}

```



```

//Example CheckBox
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class chkbox1 extends Applet implements ActionListener {
    /* Declaration */
    Label l;
    private Checkbox product1;
    private Checkbox product2;
    private Checkbox product3;
    private TextField Result;
    Button b=new Button("GetAmount");
    public chkbox1 () {
        /* Instantiation */
        l=new Label("List of Products");
        product1 = new Checkbox ("Sony HiFi");
        product2 = new Checkbox ("Sony HandyCam");
        product3 = new Checkbox ("Sony WalkMan");
        Result = new TextField ("0", 10);
        add(l);

```

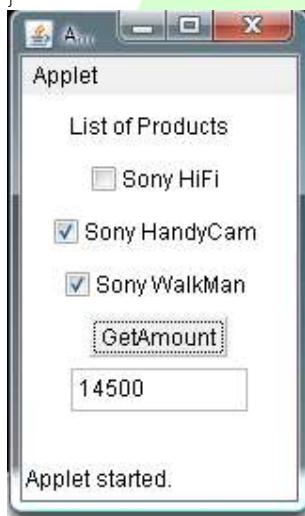
```

        add (product1);
        add (product2);
        add (product3);
    add(b);
    add (Result);

    /* Configuration */
    b.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    int amt=0;
    if (product1.getState()) {
        amt=amt+20000;
    }
    if (product2.getState()) {
        amt=amt+10000;}
    if (product3.getState()) {
        amt=amt+4500;}
    Result.setText(amt+"");
}
}

```



## Java AWT Choice Examples

### Usage:

The Choice class can be used to add a drop-down list of items to a container.

### Package:

`Java.awt`

### Subclass of:

`Java.awt.Component`

### Common Constructors:

`Choice ()`

Creates an empty Choice instance.

## Common Methods:

```
void addItemListener (ItemListener Handler)
    Configures an event handler for the choice box.
void insert (String Item, int Index)
    Inserts the item into this choice at the specified position.
int getSelectedIndex ()
    Returns the index of the selected item.
String getSelectedItem ()
    Returns the caption of the selected item.
void setBackground (Color BackgroundColor)
    Sets the background color of the choice box.
void setFont (Font TextFont)
    Sets the font for this component.
void setForeground (Color TextColor)
    Sets the color of the text for the choice box.
void Select (int Selection)
    Sets the index of the selected item.
```

## Arguments:

BackgroundColor  
The color to be used for the background of the choice box.

Index  
Position where the item will be inserted.

Item  
A String to be listed in the choice box.

Handler  
The object which handles action events from this choice box.

Selection  
The index of the selected item.

TextColor  
The color to be used for the text of the choice box.

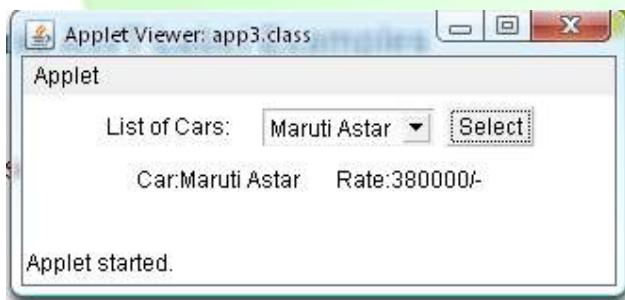
TextFont  
The font to be used for the text of the choice box.

```
//Example of Choice Class
//programme:app3.java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class app3 extends Applet implements ActionListener
{ Choice c=new Choice();
  Button b=new Button("Select");
  Label l1=new Label("Car:");
  Label l2=new Label("Rate:");
  Label l3=new Label("List of Cars:");
  public void start()
  {
    add(l3);
    c.add("Maruti Swift");
```

```

        c.add("Maruti Astar");
        c.add("Maruti 800");
        c.add("Maruti Alto");
        add(c);
        add(b);
        add(l1);
        add(l2);
        b.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    { l1.setText("Car:"+c.getSelectedItem());
        switch(c.getSelectedIndex())
        { case 0:
            l2.setText("Rate:530000/-");
            break;
        case 1:
            l2.setText("Rate:380000/-");
            break;
        case 2:
            l2.setText("Rate:200000/-");
            break;
        case 3:
            l2.setText("Rate:230000/-");
            break;
        default:
            break;
    } } }

```



## Java AWT Label Examples

### Usage:

The Label class can be used to add text labels to a container.

### Package:

`Java.awt`

### Subclass of:

`Java.awt.Component`

### Common Constructors:

`Label ()`

Creates a blank Label instance.

```
Label (String Text)
```

Creates a Label instance with the specified text.

```
Label (String Text, int Alignment)
```

Creates a Label instance with the specified text and horizontal alignment.

### **Common Methods:**

```
void setBackground (Color BgdColor)
```

Sets the color of the background for the label.

```
void setAlignment (int Alignment)
```

Sets the alignment of the label's contents along the X axis.

```
void setFont (Font TextFont)
```

Sets the font for this component.Arguments

```
void setForeground (Color TextColor)
```

Sets the color of the text for the label.

```
void setText (String Text)
```

Sets the text for the label.

### **Arguments:**

Alignment

The horizontal position of the text within the label. One of:

Label.CENTER

The central position in an area.

Label.LEFT

Box-orientation constant used to specify the left side of a box.

Label.RIGHT

Box-orientation constant used to specify the right side of a box.

BgdColor

The color to be used for the background of the label.

Text

The text to appear in the label.

TextColor

The color to be used for the text of the label.

TextFont

The font to be used for the text of the label.

## **Java AWT List Examples**

### **Usage:**

The List class can be used to add a list of items with several visible rows to a container.

### **Package:**

Java.awt

### **Subclass of:**

Java.awt.Component

### **Common Constructors:**

```
List ()
```

Creates an empty List instance with the default 4 rows visible.

```
List (int Rows)
```

Creates an empty List instance with the specified number of visible rows.

```
List (int Rows, boolean MultiSelect)
```

Creates an empty List instance with the specified number of visible rows and the specified MultiSelect switch.

### Common Methods:

`void addItemClickListener (ItemListener Handler)`

Specifies an event handler for the list.

`void add (String Item)`

Inserts the item into this choice at the next position.

`void add (String Item, int Index)`

Inserts the item into this choice at the specified position.

`int getSelectedIndex ()`

Returns the index of the selected item.

`String getSelectedItem ()`

Returns the caption of the selected item.

`void setBackground (Color BackgroundColor)`

Sets the background color of the list.

`void setFont (Font TextFont)`

Sets the font for this component.

`void setForeground (Color TextColor)`

Sets the color of the text for the list.

`void Select (int Selection)`

Sets the index of the selected item.

### Arguments:

BackgroundColor

The color to be used for the background of the list.

Index

Position where the item will be inserted.

Item

A String to be listed in the list.

Handler

The object which handles action events from this list.

Multiselect

If true, multiple items can be selected.

Selection

The index of the selected item.

TextColor

The color to be used for the text of the list.

TextFont

The font to be used for the text of the list.

### //Example of List Class

```
//program:app3.java
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

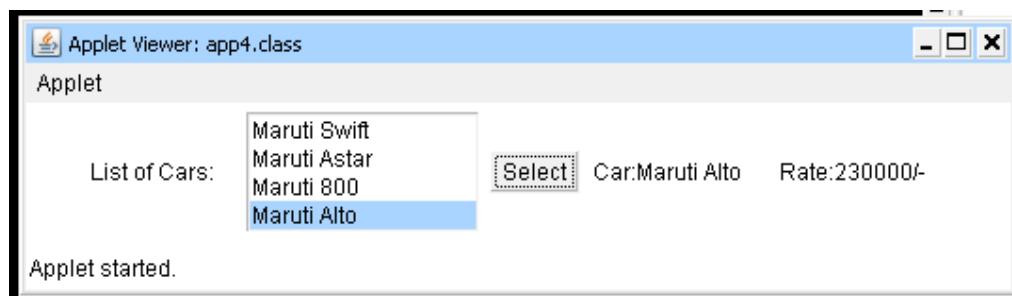
```
import java.applet.*;
```

```

public class app4 extends Applet implements ActionListener
{ List c=new List();
  Button b=new Button("Select");
  Label l1=new Label("Car:");
  Label l2=new Label("Rate:");
  Label l3=new Label("List of Cars:");
  public void start()
  {
    add(l3);
    c.add("Maruti Swift");
    c.add("Maruti Astar");
    c.add("Maruti 800");
    c.add("Maruti Alto");

    add(c);
    add(b);
    add(l1);
    add(l2);
    b.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e)
  { l1.setText("Car:"+c.getSelectedItem());
    switch(c.getSelectedIndex())
    { case 0:
        l2.setText("Rate:530000/-");
        break;
      case 1:
        l2.setText("Rate:380000/-");
        break;
      case 2:
        l2.setText("Rate:200000/-");
        break;
      case 3:
        l2.setText("Rate:230000/-");
        break;
      default:
        break; } } }

```



# Java AWT Scrollbar Examples

## Usage:

The Scrollbar lets the user graphically select a value by sliding a knob within a bounded interval.

## Package:

`Java.awt`

## Subclass of:

`Java.awt.Component`

## Common Constructors:

`Scrollbar ()`

Creates a Scrollbar instance with a range of 0-100, an initial value of 0, and vertical orientation.

`Scrollbar (int Orientation)`

Creates a Scrollbar instance with a range of 0-100, an initial value of 0, and the specified orientation.

## Common Methods:

`addAdjustmentListener (AdjustmentListener Handler)`

Configures an event handler for the scrollbar.

`getValue ()`

Returns the value of the scrollbar setting.

`setBackground (Color BackgroundColor)`

Sets the background color of the scrollbar.

`setMaximum (int Max)`

Sets the maximum value of the scrollbar.

`setMinimum (int Min)`

Sets the minimum value of the scrollbar.

`setValue (int Value)`

Sets the current value of the scrollbar.

## Arguments:

`BackgroundColor`

The color to be used for the background of the field.

`Handler`

The object which handles action events from this field.

`Max`

The maximum value of the scrollbar.

`Min`

The minimum value of the scrollbar.

`Orientation`

The orientation of the scrollbar.

`Scrollbar.HORIZONTAL`

Orients the scrollbar horizontally.

`Scrollbar.VERTICAL`

Orients the scrollbar vertically.

`Value`

The current value of the scrollbar.

## Example Code:

```
//Example Canvas,ScrollBar,BorderLayout
//Scrollbar2.java

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scrollbar2 extends Applet implements AdjustmentListener {
    /* Declaration */
    private LayoutManager Layout;
    private Scrollbar HSelector;
    private Scrollbar VSelector;
    private Drawing Pad;
    private Label Report;

    public Scrollbar2 () {
        /* Instantiation */
        Layout = new BorderLayout ();
        HSelector = new Scrollbar ();
        VSelector = new Scrollbar (Scrollbar.VERTICAL);
        Pad = new Drawing ();
        Report = new Label ();

        /* Decoration */
        HSelector.setMaximum (300);
        HSelector.setOrientation (Scrollbar.HORIZONTAL);
        VSelector.setMaximum (300);
        Report.setAlignment (Label.CENTER);
        Pad.setBackground (Color.yellow);

        /* Location */
        setLayout (Layout);
        add ("South", Report);
        add ("North", HSelector);
        add ("West", VSelector);
        add ("Center", Pad);

        /* Configuration */
        HSelector.addAdjustmentListener (this);
        VSelector.addAdjustmentListener (this);

        /* Initialization */
        HSelector.setValue (100);
        VSelector.setValue (150);
        Pad.setOval (100, 150);
        Report.setText ("H = " + HSelector.getValue() +
                        ", V = " + VSelector.getValue());
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        Report.setText ("H = " + HSelector.getValue() +
                        ", V = " + VSelector.getValue());
        Pad.setOval (HSelector.getValue (), VSelector.getValue ());
        Pad.repaint();
    }
}

class Drawing extends Canvas {
    /* Declaration */
```

```

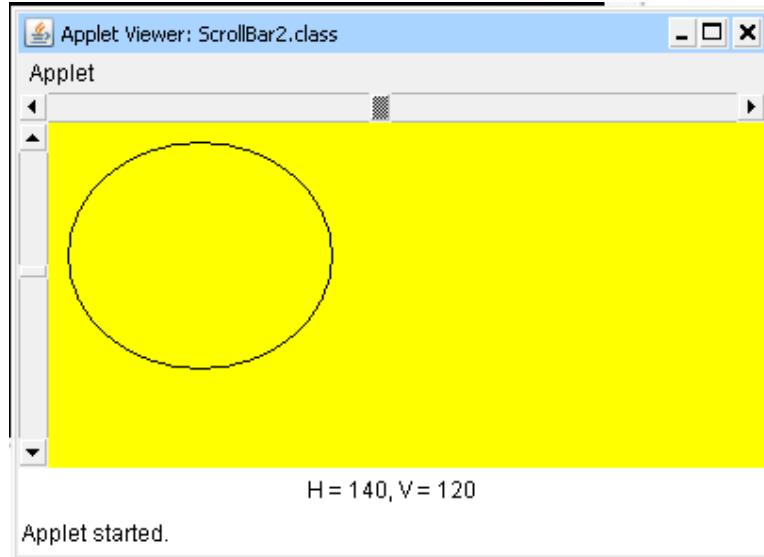
private int X;
private int Y;

public void setOval (int H, int V) {
    X = H;
    Y = V;
}

public void paint (Graphics g) {
    g.drawOval (10, 10, X, Y);
}

}

```



## Java AWT TextArea Examples

### Usage:

The `TextArea` class is a GUI component which can be used to provide multiple line text input or output.

### Package:

`java.awt`

### Subclass of:

`java.awt.TextComponent`

### Common Public Constructors:

`TextArea ()`

Creates a blank `TextArea` instance.

`TextArea (String Text)`

Creates a `TextArea` instance with the specified text.

`TextArea (int Rows, int Columns)`

Creates a blank `TextArea` instance with the specified number of rows and columns.

`TextArea (String Text, int Rows, int Columns)`

Creates a `TextArea` instance with the specified text and the specified number of rows and columns.

### Common Public Methods:

`void addTextListener (TextListener Handler);`

Configures an event handler for the `TextField`.

```
String getText ()
    Returns the text in the field.
void setBackground (Color backgroundColor)
    Sets the background color of the TextArea.
void setEditable (boolean Editable)
    Sets the field as being editable or fixed.
void setFont (Font TextFont)
    Sets the font for this component.
void setText (String Text)
    Sets the text for the field.
```

### Arguments:

BackgroundColor

The color to be used for the background of the area.

Columns

The width of the text area.

Editable

The state of the area as editable or fixed.

Handler

The object which handles action events from this combo box.

Rows

The height of the text area.

Text

The text to appear in the area.

TextFont

The font to be used for the text of the button.

### //Example TextArea

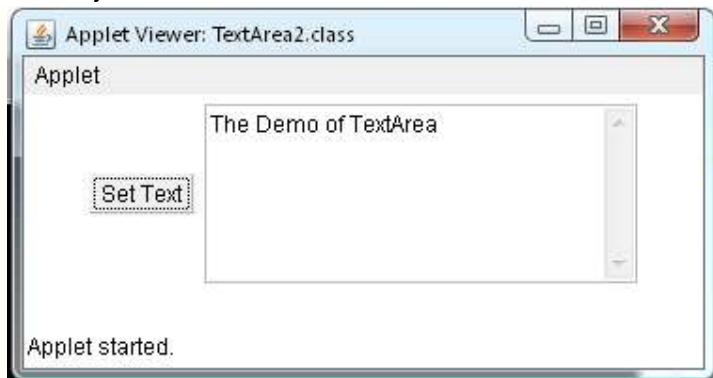
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class TextArea2 extends Applet implements ActionListener {
    /* Declaration */

    private TextArea ta;
    private Button input;
    public TextArea2 () {
        /* Instantiation */
        ta = new TextArea (5, 30);
        input=new Button("Set Text");
        /* Configuration */
        input.addActionListener (this);

        /* Location */
        add (input);
        add (ta); }
```

```
    public void actionPerformed (ActionEvent e) {  
        ta.setText ("The Demo of TextArea");  
    }  
}
```



## TextField class

The TextField class can be used to add single line text input to a container.

### Package

Java.awt

### Common Public Constructors

#### TextField ()

Creates a blank TextField instance.

#### TextField (String Text)

Creates a TextField instance with the specified text.

#### TextField (int Columns)

Creates a blank TextField instance with the specified number of columns.

#### TextField (String Text, int Columns)

Creates a TextField instance with the specified text and the specified number of columns.

### Common Public Methods

#### void addActionListener (ActionListener Handler)

Configures an event handler for the TextField.

#### String getText ()

Returns the text in the field.

#### void setBackground (Color BackgroundColor)

Sets the background color of the TextField.

```
void setEditable (boolean Editable)
    Sets the field as being editable or fixed.
void setFont (Font TextFont)
    Sets the font for this component.
void setText (String Text)
    Sets the text for the field.
```

## Arguments

### BackgroundColor

The color to be used for the background of the field.

### Editable

The state of the field as editable or fixed.

### Handler

The object which handles action events from this field.

### Text

The text to appear in the field.

### TextFont

The font to be used for the text of the button.

## Layouts

Layouts tell Java **where to put components** in containers (JPanel, content pane, etc). Every panel (and other container) has a default layout, but it's better to set the layout explicitly for clarity.

### BorderLayout

#### Java LayoutManager

#### Usage

The BorderLayout class is a Layout Manager which can be used to arrange controls in a container. BorderLayouts contain five areas: CENTER, NORTH, SOUTH, EAST, and WEST.

#### Package

Java.awt

#### Subclass of

java.lang.Object  
java.awt.BorderLayout implements LayoutManager2, Serializable

#### Common Constructors

```
BorderLayout(); // Default is no gaps
```

Creates an empty BorderLayout instance.

`BorderLayout(hgap, vgap);`

Where `hgap` and `vgap` are the distances in pixels between the regions

## Public Static Fields

### CENTER

The center layout constraint (middle of container).

### EAST

The east layout constraint (right of container).

### NORTH

The north layout constraint (top of container).

### SOUTH

The south layout constraint (bottom of container).

### WEST

The west layout constraint (left of container).

## Example Source

```
import java.applet.Applet;
import java.awt.*;

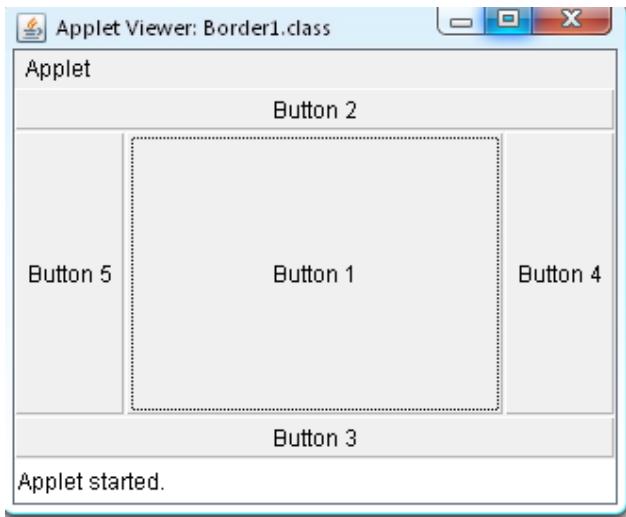
public class Border1 extends Applet {
    LayoutManager Layout;
    Button [] Buttons;

    public Border1 () {
        int i;

        Layout = new BorderLayout ();
        setLayout (Layout);

        Buttons = new Button [5];
        for (i = 0; i < 5; ++i) {
            Buttons[i] = new Button ();
            Buttons[i].setLabel ("Button " + (i + 1));
        }

        add (Buttons[0], BorderLayout.CENTER);
        add (Buttons[1], BorderLayout.NORTH);
        add (Buttons[2], BorderLayout.SOUTH);
        add (Buttons[3], BorderLayout.EAST);
        add (Buttons[4], BorderLayout.WEST);
    }
}
```



## FlowLayout

### Usage

The `FlowLayout` class is a Layout Manager which can be used to arrange controls in a container. The `FlowLayout` manager arranges items from left to right, in a line, going to the next line when the line fills up.

`java.awt.FlowLayout` arranges components from left-to-right and top-to-bottom, centering components horizontally with a five pixel gap between them. When a container size is changed (eg, when a window is resized), `FlowLayout` recomputes new positions for all components subject to these constraints.

Use `FlowLayout` because it's quick and easy. It's a good first choice when using *iterative* development. I often start with a `FlowLayout` in an early iteration then switch to a better layout, if necessary, on a later iteration.

### Package

`Java.awt`

### Subclass of

`java.lang.Object`

`java.awt.FlowLayout` implements `LayoutManager`, `Serializable`

### Common Constructors

Typically the constructor is called in the call to the container's `setLayout` method (see example code). The parameterless `FlowLayout()` constructor is probably most common, but there are some good places to use the alignment.

```
new FlowLayout()           // default is centered with 5 pixel gaps  
new FlowLayout(int align)  
new FlowLayout(int align, int hgap, int vgap)
```

## Alignment

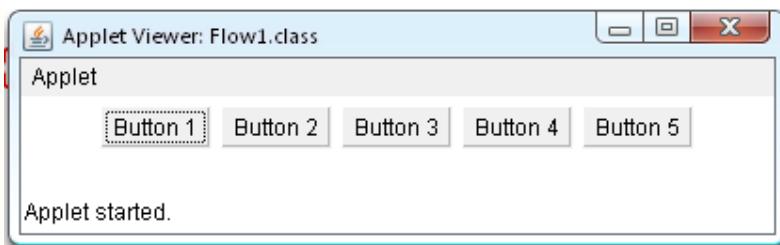
`align` is one of `FlowLayout.LEFT`, `FlowLayout.CENTER` (the default), or `FlowLayout.RIGHT`. You might want to use the `RIGHT` alignment when building a dialog that puts the `OK` and `Cancel` buttons at the lower right.

## Spacing

The default spacing is good for most purposes and is rarely changed. `hgap` is the size in pixels of the horizontal gap (distance) between components, and `vgap` is the vertical gap.

## Example Code:

```
import java.applet.Applet;  
import java.awt.*;  
  
public class Flow1 extends Applet {  
    LayoutManager Layout;  
    Button [] Buttons;  
  
    public Flow1 () {  
        int i;  
  
        Layout = new FlowLayout ();  
        setLayout (Layout);  
  
        Buttons = new Button [5];  
        for (i = 0; i < 5; ++i) {  
            Buttons[i] = new Button ();  
            Buttons[i].setLabel ("Button " + (i + 1));  
            add (Buttons[i]);  
        }  
    }  
}
```



## GridLayout

### Usage

The GridLayout class is a Layout Manager which can be used to arrange controls in a container. GridLayouts have a specified number of rows and columns.

### Package

Java.awt

### Subclass of

java.lang.Object

java.awt.FlowLayout implements LayoutManager, Serializable

### Common Constructors

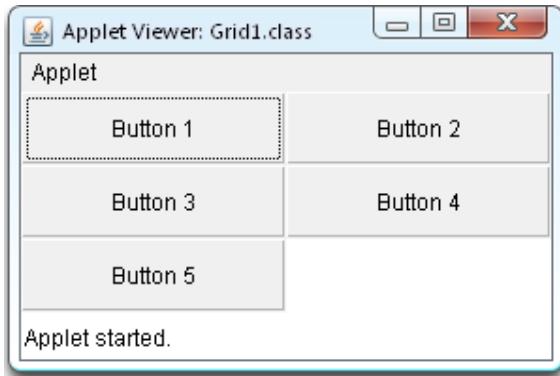
There are three constructors:

```
GridLayout() // One row. Columns expand.  
GridLayout(rows, cols);  
GridLayout(rows, cols, hgap, vgap)
```

with the following int parameters: *rows* is number of rows, *cols* is number of columns, *hgap* is horizontal space between components (in pixels), and *vgap* is vertical space between components (in pixels).

### Example Code:

```
import java.applet.Applet;  
import java.awt.*;  
  
public class Grid1 extends Applet {  
    LayoutManager Layout;  
    Button [] Buttons;  
  
    public Grid1 () {  
        int i;  
  
        Layout = new GridLayout (3, 2);  
        setLayout (Layout);  
  
        Buttons = new Button [5];  
        for (i = 0; i < 5; ++i) {  
            Buttons[i] = new Button ();  
            Buttons[i].setLabel ("Button " + (i + 1));  
            add (Buttons[i]);  
        }  
    }  
}
```



## **CardLayout**

A `CardLayout` object is a layout manager for a container. It treats each component in the container as a card. Only one card is visible at a time, and the container acts as a stack of cards. The first component added to a `CardLayout` object is the visible component when the container is first displayed.

The ordering of cards is determined by the container's own internal ordering of its component objects. `CardLayout` defines a set of methods that allow an application to flip through these cards sequentially, or to show a specified card.

## **Constructors**

```
public CardLayout()
```

Creates a new card layout with gaps of size zero.

```
public CardLayout(int hgap,  
                  int vgap)
```

Creates a new card layout with the specified horizontal and vertical gaps. The horizontal gaps are placed at the left and right edges. The vertical gaps are placed at the top and bottom edges.

## **Methods Of CardLayout**

### **addLayoutComponent**

```
public void addLayoutComponent(String name,  
                               Component comp)
```

**Deprecated.** replaced by `addLayoutComponent(Component, Object)`.

**Description copied from interface:** `LayoutManager`

If the layout manager uses a per-component string, adds the component `comp` to the layout, associating it with the string specified by `name`.

### **removeLayoutComponent**

```
public void removeLayoutComponent(Component comp)
```

Removes the specified component from the layout.

## **first**

```
public void first(Container parent)
    Flips to the first card of the container.
```

### **Parameters:**

parent - the name of the parent container in which to do the layout.

## **next**

```
public void next(Container parent)
    Flips to the next card of the specified container. If the currently visible card is the last one,
    this method flips to the first card in the layout.
```

### **Parameters:**

parent - the name of the parent container in which to do the layout.

## **previous**

```
public void previous(Container parent)
    Flips to the previous card of the specified container. If the currently visible card is the first
    one, this method flips to the last card in the layout.
```

### **Parameters:**

parent - the name of the parent container in which to do the layout.

## **last**

```
public void last(Container parent)
    Flips to the last card of the container.
```

### **Parameters:**

parent - the name of the parent container in which to do the layout.

## **show**

```
public void show(Container parent,
                  String name)
```

Flips to the component that was added to this layout with the specified name, using addLayoutComponent. If no such component exists, then nothing happens.

### **Parameters:**

parent - the name of the parent container in which to do the layout.

name - the component name.

```
//cardlayout
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class card extends Applet implements ActionListener
{ //Panel first for electronics product
    Panel F=new Panel();
    Checkbox c1=new Checkbox("LCD");
    Checkbox c2=new Checkbox("HiFi");
    Button B1=new Button("Electronics");
//Panel Second for Computer Hardware
    Panel S=new Panel();
    Checkbox c3=new Checkbox("DVD");
    Checkbox c4=new Checkbox("RAM");
    Button B2=new Button("Computer Hardware");

    //Panel Third for Electrical Product
    Panel T=new Panel();
    Checkbox c5=new Checkbox("Iron");
    Checkbox c6=new Checkbox("Mixer");
    Button B3=new Button("Electrical");

    //Main Panel
    Panel MAIN=new Panel();
    CardLayout c=new CardLayout();
    public void start()
    { MAIN.setLayout(c);
        add(B1);
        add(B2);
        add(B3);
        add(MAIN);
        // add items in electronics panel
        F.add(c1);
        F.add(c2);
        // add items in Computer Hardware panel
        S.add(c3);
        S.add(c4);
```

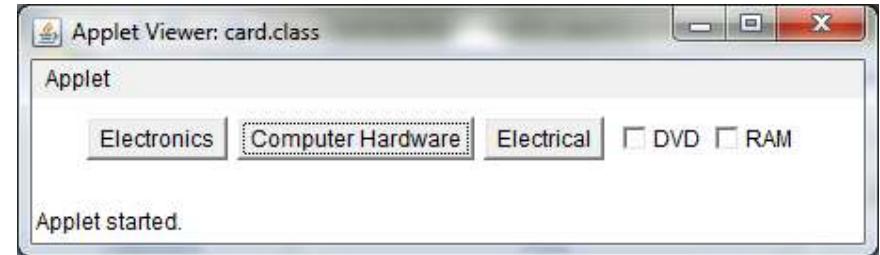
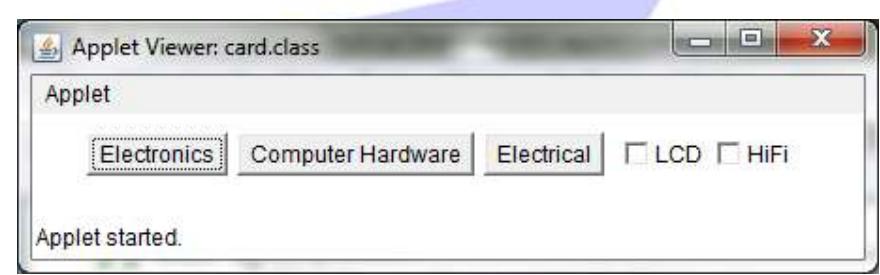
```
// add items in Electrical panel
    T.add(c5);
    T.add(c6);

    MAIN.add(F,"ELTRS");
    MAIN.add(S,"CH");
    MAIN.add(T,"ELEC");
    B1.addActionListener(this);
    B2.addActionListener(this);
    B3.addActionListener(this);

}

public void actionPerformed(ActionEvent e)
{ if(e.getSource()==B1)
    {c.show(MAIN,"ELTRS");

}
else if(e.getSource()==B2)
{c.show(MAIN,"CH");
}
else if(e.getSource()==B3)
{c.show(MAIN,"ELEC");
}}}
```



## **Menus in Applets**

### **TheMenuBar Class**

The `MenuBar` class represents the top-most level of a menu. After creating a menubar, you can create `Menu` objects and assign the `Menu` objects to the menubar. A menubar is created as follows:

```
MenuBar bar = newMenuBar();
```

In addition to its constructor, the `MenuBar` class includes the methods listed in Table 6.5.

**Table 6.5. Nonprivate methods of theMenuBar class.**

| <i>Method</i>                      | <i>Purpose</i>                                          |
|------------------------------------|---------------------------------------------------------|
| <code>add(Menu)</code>             | Adds a menu to this menubar.                            |
| <code>countMenus()</code>          | Returns the number of menus on this menubar.            |
| <code>getMenu(int)</code>          | Returns the menu at the specified index.                |
| <code>remove(int)</code>           | Removes the menu at the specified index.                |
| <code>remove(MenuComponent)</code> | Removes the specified menu component from this menubar. |

In Java, menus are added to a class that implements the `MenuContainer` interface. In most cases this will be a class you define and base on `Frame` because `Frame` is the only Java container to implement `MenuContainer`. The following code adds a menubar to a frame:

```
MenuBar bar = newMenuBar();
// add menus to theMenuBar
myFrame.setMenuBar(bar);
```

### **The Menu Class**

A menubar without any menus is as worthless as a pizza without any pepperoni. This can be easily rectified by creating `Menu` objects. To create a new menu, simply provide the name of the menu to the constructor, as follows:

```
Menu drw= new Menu("Drawing");
```

After creating a menu, you must add it to the menubar. Menus will be displayed across the menubar in the order they are added to it. For example, the following code will create a Drawing menu:

```
Menu drw= new Menu("Drawing");
```

```
menuBar.add(drw);
```

The `Menu` class includes the methods listed in Table.

**Table Nonprivate methods of the Menu class.**

| Method                             | Purpose                                              |
|------------------------------------|------------------------------------------------------|
| <code>add(MenuItem)</code>         | Adds the specified menu item to the menu.            |
| <code>add(String)</code>           | Adds an item with the specified label to the menu.   |
| <code>addSeparator()</code>        | Adds a separator line to the menu.                   |
| <code>countItems()</code>          | Returns the number of items in the menu.             |
| <code>getItem(int)</code>          | Returns the menu item at the specified index.        |
| <code>remove(int)</code>           | Removes the menu item at the specified index.        |
| <code>remove(MenuComponent)</code> | Removes the specified menu component from the menu . |

## The MenuItem Class

The menu still isn't useful, however, because no menu items have been added. A new menu item can be created by passing a string to the menu item constructor. For example, the following code illustrates all that is needed to create a menubar, a Dogs menu on the menubar, and three items on the Dogs menu:

```
MenuBar mb=newMenuBar();
    setMenuBar(mb);
    Menu drw= new Menu("Drawing");
    MenuItem i1,i2,i3,i4;
    MenuShortcut M=new MenuShortcut(65);
    i1=new MenuItem("Line",M);
    i2=new MenuItem("Rectangle");
    i4=new MenuItem("Oval");
    i3=new MenuItem("-");
    drw.add(i1);
    drw.add(i2);
    drw.add(i3);
    drw.add(i4);
    mb.add(drw);
```

By default, `MenuItem`s are *enabled*. This means that the user can select them from the menu on which they appear. It is possible to disable a `MenuItem`. This can be done using either the `disable` method or by passing `false` to the `enable` method. A `MenuItem` can be enabled by passing `true` to `enable` or by using `enable` without any parameters. These methods are illustrated in the following:

```
MenuItem i1 = new MenuItem("Line");
i1.disable();           // disable the MenuItem
```

The `enable` and `disable` methods are not the only ones available for a `MenuItem`. Table describes each of the nonprivate members of `MenuItem`.

**Table Nonprivate methods of the MenuItem class.**

| Method                        | Purpose                                                                        |
|-------------------------------|--------------------------------------------------------------------------------|
| <code>disable()</code>        | Disables selection of this menu item.                                          |
| <code>enable()</code>         | Enables selection of this menu item.                                           |
| <code>enable(boolean)</code>  | Enables or disables selection of the menu item based on the specified Boolean. |
| <code>getLabel()</code>       | Returns the label for this menu item.                                          |
| <code>IsEnabled()</code>      | Returns <code>true</code> if the menu item is selectable.                      |
| <code>setLabel(String)</code> | Sets the label of the menu item to the specified string.                       |

## //Example of Menu

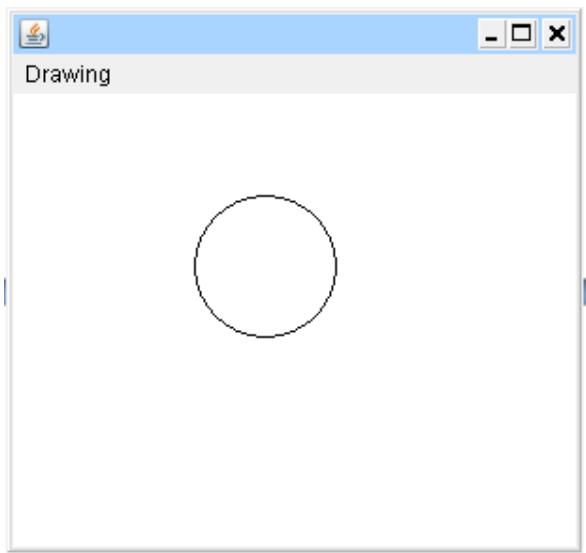
```
//Example of Menu
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class mnu extends Frame
{int s=0;
    mnu()
    {
        MenuBar mb=new MenuBar();
        setMenuBar(mb);
        Menu drw= new Menu("Drawing");
        MenuItem i1,i2,i3,i4;
        MenuShortcut M=new MenuShortcut(65);
        i1=new MenuItem("Line",M);
        i2=new MenuItem("Rectangle");
        i4=new MenuItem("Oval");
        i3=new MenuItem("-");
        drw.add(i1);
        drw.add(i2);
        drw.add(i3);
        drw.add(i4);
        mb.add(drw);
        i1.addActionListener(new hnd());
        i2.addActionListener(new hnd());
        i4.addActionListener(new hnd());
        setSize(300,300);
        setVisible(true);
```

```
}

public void paint(Graphics g)
{ switch(s)
{case 1:
    g.drawLine(100,100,200,200);
    break;
case 2:
    g.drawRect(100,100,75,75);
    break;
case 3:
    g.drawOval(100,100,75,75);
    break;
}
}

class hnd implements ActionListener
{ public void actionPerformed(ActionEvent e)
{ String n=(String)e.getActionCommand();
//dlg d=new dlg(new mnu(),"First Dialog");
if (n.equals("Line"))
{
s=1;
repaint();
}
else if (n.equals("Rectangle"))
{
s=2;
repaint();
}
else if (n.equals("Oval"))
{
s=3;
repaint();
}
}
}

public class menu
{ public static void main(String arg[])
{new mnu();}}
```





# **Swings with Examples**

# Java javax.swing Package Examples

## Usage

The JApplet class can be used to create web page applets.

## Package

javax.swing

## Class

javax.swing.JApplet

## Parent Classes

javax.swing.JApplet  
java.applet.Applet  
java.awt.Panel  
java.awt.Container  
java.awt.Component  
java.lang.Object

## Common Public Constructors

**public JApplet ()**

Creates a blank JApplet instance.

## Common Public Methods

**Container getContentPane ()**

Returns the contentPane object for this applet.

**int getHeight ()**

Returns the vertical size of the applet.

**int getWidth ()**

Returns the horizontal size of the applet.

**void init ()**

The *init* method is called once after the constructor is executed.

**void paint (Graphics g)**

The *paint* method is called every time the the applet is to be redrawn.

**void start ()**

The *start* method is called when the web page or appletviewer is redisplayed.

**void setBackground (Color BgColor)**

Sets the background color of the text for the applet.Arguments  
**void setForeground (Color FgColor)**  
Sets the foreground color of the applet.

## Arguments

### BgColor

The color to be used for the background of the applet.

### FgColor

The color to be used for the foreground of the applet.

### g

The Graphics object which can be used for drawing in the applet.

## Applet - Applet2

### Source Code - Applet2.java

```
/*
 * File:          Applet2.java
 * Description:  Simple swing JApplet
 * Date:         11/21/08
 */

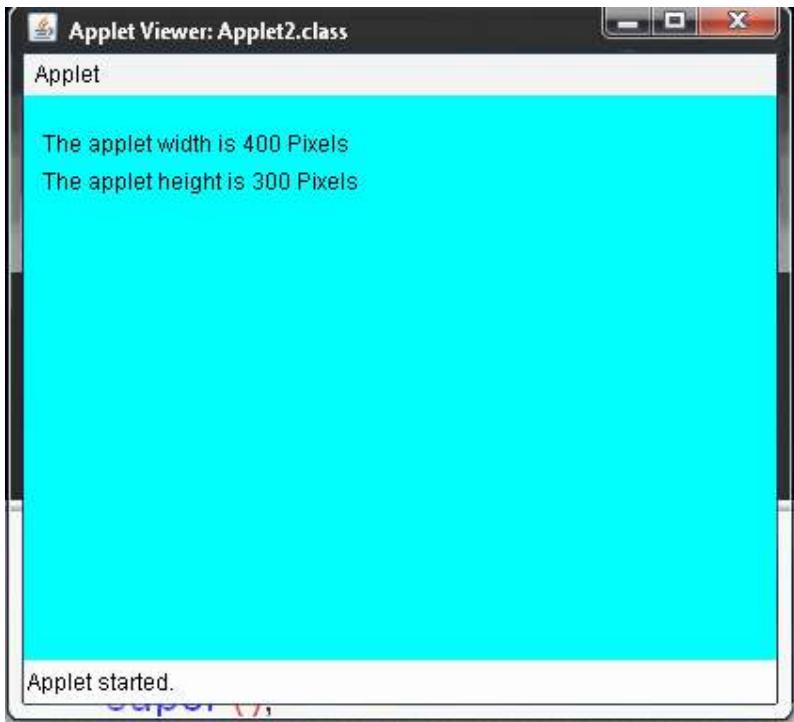
import java.awt.*;
import javax.swing.*;

public class Applet2 extends JApplet {
    private Container Panel;

    public Applet2 () {
        super ();
        Panel = getContentPane();
        Panel.setBackground (Color.cyan);
    }

    public void paint (Graphics g) {
        int Width;
        int Height;

        super.paint (g);
        Width = getWidth();
        Height = getHeight();
        g.drawString ("The applet width is " + Width + " Pixels", 10, 30);
        g.drawString ("The applet height is " + Height + " Pixels", 10, 50);
    }
}
```



## **JFrame**

### **Usage**

A JFrame is a top-level window with a title and a border. It is normally used as the main window of a standalone application.

### **Package**

Java.swing

### **Class**

Java.swing.JFrame

### **Parent Classes**

Java.swing.JFrame implements WindowConstants, Accessible, RootPaneContainer

java.awt.Frame implements MenuContainer

java.awt.Window implements Accessible

java.awt.Container

java.awt.Component implements ImageObserver, MenuContainer, Serializable

java.lang.Object

### **Common Public Constructors**

**JFrame ()**

Creates a blank Button instance.

#### **JFrame (String Text)**

Creates a Frame instance with the specified text in the title bar.

## Common Public Methods

### **void addWindowListener(WindowListener Handler)**

Configures a window event handler for the frame.

### **Container getContentPane()**

Returns the contentPane object for this frame.

### **void setBackground (Color BackgroundColor)**

Sets the background color of the frame.

### **void setFont (Font TextFont)**

Sets the font for this component.

### **void setForeground (Color TextColor)**

Sets the color of the text for the frame.

### **void setSize (int Width, int Height)**

Resizes this window so that it has the specified Width and Height.

### **void setTitle (String Text)**

Sets the text for the title bar.

### **void show ()**

Makes the window visible.

## Arguments

### **BackgroundColor**

The color to be used for the background of the button.

### **Handler**

The object which handles window events from this frame.

### **Text**

The text to appear in the title bar.

### **TextColor**

The color to be used for the text of the button.

### **TextFont**

The font to be used for the text of the button.

## Example - Frame1.class



## Source Code - Frame1.java

```
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;

public class Frame1 extends JFrame {
    private Closer Handler;
    private JLabel Tag;

    Frame1 () {
        Container Pane;
        Handler = new Closer ();
        Tag = new JLabel ();
        Pane = getContentPane ();
        setTitle ("JFrame Example");
        setSize (300,120);
        Tag.setText ("JLabel");
        Pane.add (Tag);
        addWindowListener (Handler);
        show ();
    }

    public static void main (String args[]) {
        JFrame f;
        f = new Frame1 ();
    }
}

class Closer extends WindowAdapter {
    public void windowClosing (WindowEvent event) {
        System.exit (0);
    }
}
```

## JPanel

### Usage

JPanel is a generic lightweight container: a generic Swing container object that can contain other Swing and AWT components. Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified when adding a component to a container, it will be added to the end of the list (and hence to the bottom of the stacking order).

### Package

javax.swing

### Class

javax.swing.JPanel

### Parent Classes

javax.swing.JPanel implements Accessible

```
javax.swing.JComponent implements Serializable  
java.awt.Container  
java.awt.Component implements ImageObserver, MenuContainer, Serializable  
java.lang.Object
```

## Common Public Constructors

### JPanel ()

Constructs a new blank JPanel.

## Common Public Methods

### void paint (Graphics g)

Overrides the paint method of JPanel.

### void repaint ()

Repaints this component, and causes a call to the paint method.

### void setLayout (LayoutManager Manager)

Sets the layout manager for the panel.

### Component add (Component Item)

Appends the specified component to the end of this container.

### void add (Component Item, Object Constraints)

Appends the specified component with the specified constraints.

## Arguments

### g

A Graphics object on which the paint method can draw.

### item

A component to be added to the panel.

### Manager

A LayoutManager which will arrange the components of the panel.

## Applet - JPanel1.class

## Source Code - JPanel1.java

```
import java.awt.event.*;  
import java.awt.*;  
import javax.swing.*;  
import java.awt.geom.*;  
  
public class JPanel1 extends JApplet  
    implements ActionListener {  
    JPanel InputPanel;  
    JLabel Tag;  
    JTextField Entry;  
    CircleCanvas Drawing;
```

```

public JPanel1 () {
    LayoutManager Layout;
    Container Pane;

    Layout = new FlowLayout ();
    InputPanel = new JPanel ();
    Tag = new JLabel ();
    Entry = new JTextField ();
    Drawing = new CircleCanvas ();
    Pane = getContentPane ();

    InputPanel.setLayout (Layout);
    Tag.setText ("Diameter ");
    Tag.setHorizontalAlignment (JLabel.RIGHT);
    Entry.setColumns (8);
    Entry.setText ("10");
    Entry.setHorizontalAlignment (JTextField.RIGHT);
    Entry.addActionListener (this);
    InputPanel.add (Tag);
    InputPanel.add (Entry);
    Pane.add (InputPanel, BorderLayout.NORTH);
    Pane.add (Drawing, BorderLayout.CENTER);
}

public void actionPerformed (ActionEvent e) {
    int Diameter;
    String Text;

    Text = Entry.getText ();
    Diameter = Integer.parseInt (Text);
    Drawing.setDiameter (Diameter);
    Drawing.repaint ();
}

class CircleCanvas extends Canvas {
    int Diameter;

    public CircleCanvas () {
        setBackground (Color.yellow);
    }

    public void setDiameter (int Value) {
        Diameter = Value;
        repaint ();
    }

    public void paint (Graphics g) {
        Graphics2D g2;
        Ellipse2D Circle;
        int Width;
        int Height;

        g2 = (Graphics2D) g;
        g2.setColor (Color.black);
        Width = getWidth ();
        Height = getHeight ();
        Circle = new Ellipse2D.Double
            ((double) ((Width - Diameter) / 2),
             (double) ((Height - Diameter) / 2),

```

```
        (double) Diameter, (double)Diameter);
    g2.fill (Circle);
}

}
```

## Java Swing Examples

### JButton

#### Abstract

Describes the Java Swing JButton class, with commonly used methods, and uses a JButton in a live applet example, with source code.

#### Usage

The JButton class can be used to add interactive buttons to a container.

#### Package

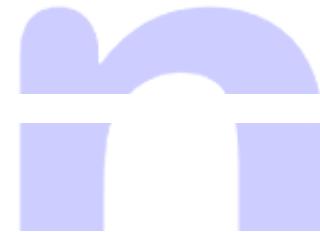
javax.swing

#### Class

javax.swing.JButton

#### Parent Classes

javax.swing.JButton implements Accessible  
javax.swing.AbstractButton implements ItemSelectable, SwingConstants  
javax.swing.JComponent  
java.awt.Container  
java.awt.Component implements ImageObserver, MenuContainer, Serializable  
java.lang.Object



#### Common Constructors

##### **JButton ()**

Creates a blank JButton instance.

##### **JButton (String Text)**

Creates a JButton instance with the specified text.

#### Common Methods

##### **void addActionListener (ActionListener Handler)**

Configures an event handler for the button.

##### **void setActionCommand (String ActionText)**

Sets the text for the action event of the button.

##### **void setBackground (Color BackgroundColor)**

Sets the background color of the button.  
**void setEnabled (boolean State)**  
Enables or disables the button.  
**void setFont (Font TextFont)**  
Sets the font for this component.  
**void setForeground (Color TextColor)**  
Sets the color of the text for the button.  
**void setText (String Text)**  
Sets the text for the button.

## Arguments

### ActionText

The ActionCommand text for the event associated with the button.

### BackgroundColor

The color to be used for the background of the button.

### Handler

The object which handles action events from this combo box.

### State

The enable state of the button (true or false.)

### Text

The text to appear in the button.

### TextColor

The color to be used for the text of the button.

### TextFont

The font to be used for the text of the button.

## Applet - Button1.class

## Source Code - Button1.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Button1 extends JApplet implements ActionListener {
    /* Declaration */
    private Container Panel;
    private LayoutManager Layout;
    private JButton Button1;
    private JButton Button2;
    private JButton Button3;
    private JTextField Command;
```

```

public Button1 () {
    /* Instantiation */
    Layout = new FlowLayout ();
    Panel = getContentPane ();
    Button1 = new JButton ("Red Background");
    Button2 = new JButton ("Yellow Background");
    Button3 = new JButton ();
    Command = new JTextField ("ActionCommand", 20);

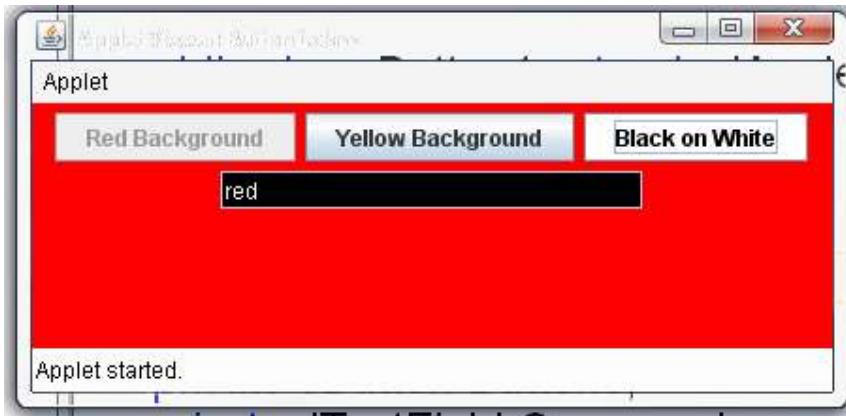
    /* Location */
    Panel.setLayout (Layout);
    Panel.add (Button1);
    Panel.add (Button2);
    Panel.add (Button3);
    Panel.add (Command);

    /* Configuration */
    Button1.addActionListener (this);
    Button1.setActionCommand ("red");
    Button2.setEnabled (false);
    Button2.addActionListener (this);
    Button3.addActionListener (this);
    Command.setEditable (false);

    /* Decoration */
    Button3.setBackground (Color.black);
    Button3.setForeground (Color.white);
    Button3.setText ("White on Black");
    Panel.setBackground (Color.yellow);
}

public void actionPerformed(ActionEvent e) {
    String Action;
    Action = e.getActionCommand ();
    if (Action.equals ("red")) {
        Panel.setBackground (Color.red);
        Button1.setEnabled (false);
        Button2.setEnabled (true);
    } else if (Action.equals ("Yellow Background")) {
        Panel.setBackground (Color.yellow);
        Button1.setEnabled (true);
        Button2.setEnabled (false);
    } else if (Action.equals ("White on Black")) {
        Command.setBackground (Color.black);
        Command.setForeground (Color.white);
        Button3.setBackground (Color.white);
        Button3.setForeground (Color.black);
        Button3.setText ("Black on White");
    } else if (Action.equals ("Black on White")) {
        Command.setBackground (Color.white);
        Command.setForeground (Color.black);
        Button3.setBackground (Color.black);
        Button3.setForeground (Color.white);
        Button3.setText ("White on Black");
    }
    Command.setText (Action);
}
}

```



## JCheckBox

### Usage

The JCheckBox class can be used to add independent check boxes to a container.

### Package

Java.swing

### Class

Java.swing.JCheckBox

### Parent Classes

Java.swing.JCheckBox implements Accessible

javax.swing.JToggleButton implements Accessible

javax.swing.AbstractButton implements ItemSelectable, SwingConstants

javax.swing.JComponent implements Serializable

java.awt.Container

java.awt.Component implements ImageObserver, MenuContainer, Serializable

java.lang.Object

### Common Public Constructors

#### JCheckBox ()

Creates a blank JCheckBox instance.

#### JCheckBox (String Text)

Creates a JCheckBox instance with the specified text.

#### JCheckBox (String Text, boolean Selected)

Creates a JCheckBox instance with the specified text and the designated selection state.

## Common Public Methods

**void addItemListener (ItemListener Handler)**

Configures an event handler for the checkbox.

**boolean isSelected ()**

Returns the state (checked or not checked) of the checkbox.

**void setBackground (Color BackgroundColor)**

Sets the background color of the checkbox.

**void setSelected (boolean State)**

Sets the state (checked or not checked) of the checkbox.

**void setFont (Font TextFont)**

Sets the font for this component.

**void setForeground (Color TextColor)**

Sets the color of the text for the checkbox.

**void setText (String Text)**

Sets the text for the button.

## Arguments

**BackgroundColor**

The color to be used for the background of the button.

**Handler**

The object which handles action events from this combo box.

**State**

The checked state of the checkbox (true or false.)

**Text**

The text to appear in the button.

**TextColor**

The color to be used for the text of the button.

**TextFont**

The font to be used for the text of the button.

## Applet - CheckBox1.class

## Source Code - CheckBox1.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CheckBox1 extends JApplet implements ItemListener {
    /* Declaration */
    private Container Panel;
    private LayoutManager Layout;
    private JCheckBox Bold;
    private JCheckBox Italic;
    private JTextField Result;
```

```

public CheckBox1 () {
    /* Instantiation */
    Layout = new FlowLayout ();
    Bold = new JCheckBox ("Bold");
    Italic = new JCheckBox ("Italic", true);
    Result = new JTextField ("", 25);

    Panel = getContentPane ();

    /* Location */
    Panel.setLayout (Layout);
    Panel.add (Bold);
    Panel.add (Italic);
    Panel.add (Result);
    Panel.setBackground (Color.yellow);

    /* Configuration */
    Bold.addItemListener (this);
    Italic.addItemListener (this);
    Result.setEditable (false);

    /* Decoration */
    Bold.setBackground (Color.yellow);
    Italic.setBackground (Color.yellow);

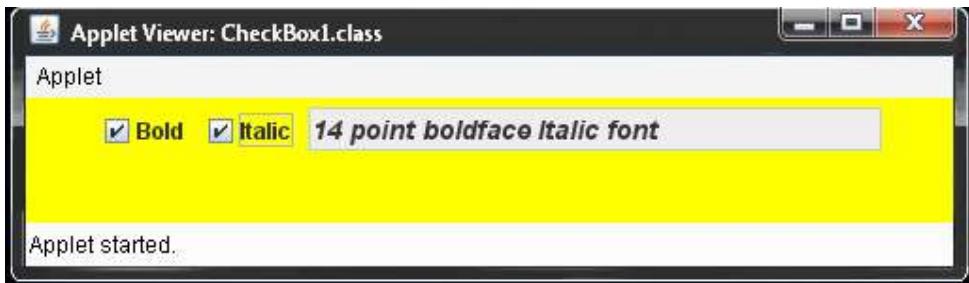
    /* Initialization */
    setState ();
}

private void setState () {
    String Text;
    int Style;
    Font ShowFont;

    Text = "14 point";
    Style = 0;
    if (Bold.isSelected()) {
        Style = Style | Font.BOLD;
        Text += " boldface";
    } else {
        Text += " regular weight";
    }
    if (Italic.isSelected()) {
        Style = Style | Font.ITALIC;
        Text += " Italic";
    } else {
        Text += " Roman";
    }
    Text += " font";
    ShowFont = new Font ("SansSerif", Style, 14);
    Result.setFont (ShowFont);
    Result.setText (Text);
}

public void itemStateChanged(ItemEvent e) {
    setState ();
}

```



## JColorChooser

### Usage

The JColorChooser class can be used to add interactive color choice to a container.

### Package

Java.awt

### Class

Java.awt.JColorChooser

### Parent Classes

Java.awt.JColorChooser implements Accessible

java.awt.Component implements Serializable

java.awt.Container

java.awt.Component implements ImageObserver, MenuContainer, Serializable

java.lang.Object

### Common Public Constructors

#### JColorChooser ()

Creates a JColorChooser instance with an initial selection of white.

#### JColorChooser (Color InitialSelection)

Creates a JColorChooser instance with a designated initial selection.

### Common Public Methods

#### void addItemListener (ItemListener Handler)

Configures an event handler for the combo box.

#### Color getColor ()

Returns the selected color.

#### ColorSelectionModel getSelectionModel ()

Returns the selection model of the JColorChooser. Note: A ChangeListener can be added to the ColorSelectionModel. See the example below.

## **void setColor (Color C)**

Sets the current color of the color chooser to the specified color.

## **Arguments**

### **InitialSelection**

The color to be used as the initial selection.

### **Handler**

The object which handles action events from this color chooser.

### **C**

The color to be used for the operation.

## **Applet - ColorChooser1.class**

## **Source Code - ColorChooser1.java**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.colorchooser.*;
import javax.swing.event.*;

public class ColorChooser1 extends JApplet implements ChangeListener {
    /* Declaration */
    private Container Panel;
    private LayoutManager Layout;
    private JColorChooser Chooser;
    private ColorSelectionModel Model;

    public ColorChooser1 () {

        /* Instantiation */
        Layout = new FlowLayout ();
        Chooser = new JColorChooser ();
        Model = Chooser.getSelectionModel();
        Panel = getContentPane ();

        /* Location */
        Panel.setLayout (Layout);
        Panel.add (Chooser);

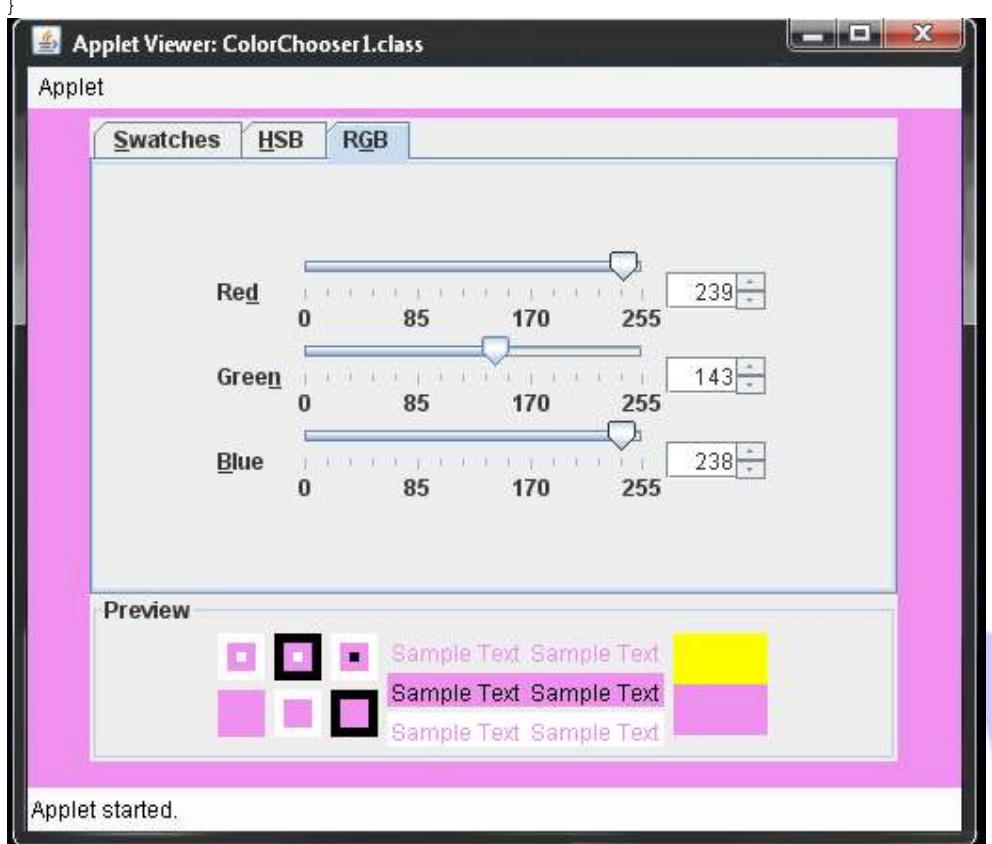
        /* Decoration */
        Chooser.setColor (Color.yellow);
        Panel.setBackground (Color.yellow);

        /* Configuration */
        Model.addChangeListener (this);
    }

    public void stateChanged(ChangeEvent e) {
        Color Choice;
```

```
        Choice = Chooser.getColor();
        Panel.setBackground (Choice);
    }

}
```



## JComboBox

### Abstract

Describes the Java Swing JComboBox class, with commonly used methods, and uses a JComboBox in a live applet example, with source code.

### Usage

The JComboBox class can be used to add a drop-down list of items to a container.

### Package

javax.swing

### Class

javax.swing.JComboBox

### Parent Classes

```
javax.swing.JComboBox implements Accessible, ActionListener, EventListener,  
ItemSelectable, ListDataListener  
javax.swing.JComponent  
java.awt.Container  
java.awt.Component implements ImageObserver, MenuContainer, Serializable  
java.lang.Object
```

## Common Constructors

### **JComboBox ()**

Creates an empty JComboBox instance.

### **JComboBox (Object [] Items)**

Creates a JComboBox instance that contains the elements in the specified array.

## Common Methods

### **void addItemListener (ItemListener Handler)**

Configures an event handler for the combo box.

### **int getSelectedIndex ()**

Returns the index of the selected item.

### **void setBackground (Color BackgroundColor)**

Sets the background color of the combo box.

### **void setFont (Font TextFont)**

Sets the font for this component.

### **void setForeground (Color TextColor)**

Sets the color of the text for the combo box.

### **void setSelectedIndex (int Selection)**

Sets the index of the selected item.

## Arguments

### **BackgroundColor**

The color to be used for the background of the combo box.

### **Items**

An array of objects to be listed in the combo box.

### **Handler**

The object which handles action events from this combo box.

### **Selection**

The index of the selected item.

### **TextColor**

The color to be used for the text of the combo box.

### **TextFont**

The font to be used for the text of the combo box.

## Source Code - ComboBox1.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComboBox1 extends JApplet implements ItemListener {
    private Container Panel;
    private LayoutManager Layout;
    private JComboBox Selector;
    private String [] ColorList;
    private Font SansSerif;

    public ComboBox1 () {
        /* Instantiation */
        Layout = new FlowLayout ();
        ColorList = new String [9];
        SansSerif = new Font ("SansSerif", Font.BOLD, 14);

        ColorList [0] = "Red";
        ColorList [1] = "Magenta";
        ColorList [2] = "Blue";
        ColorList [3] = "Cyan";
        ColorList [4] = "Green";
        ColorList [5] = "Yellow";
        ColorList [6] = "White";
        ColorList [7] = "Gray";
        ColorList [8] = "Black";

        Selector = new JComboBox (ColorList);

        /* Location */
        Panel = getContentPane ();
        Panel.setLayout (Layout);
        Panel.add (Selector);
        Panel.setBackground (Color.yellow);

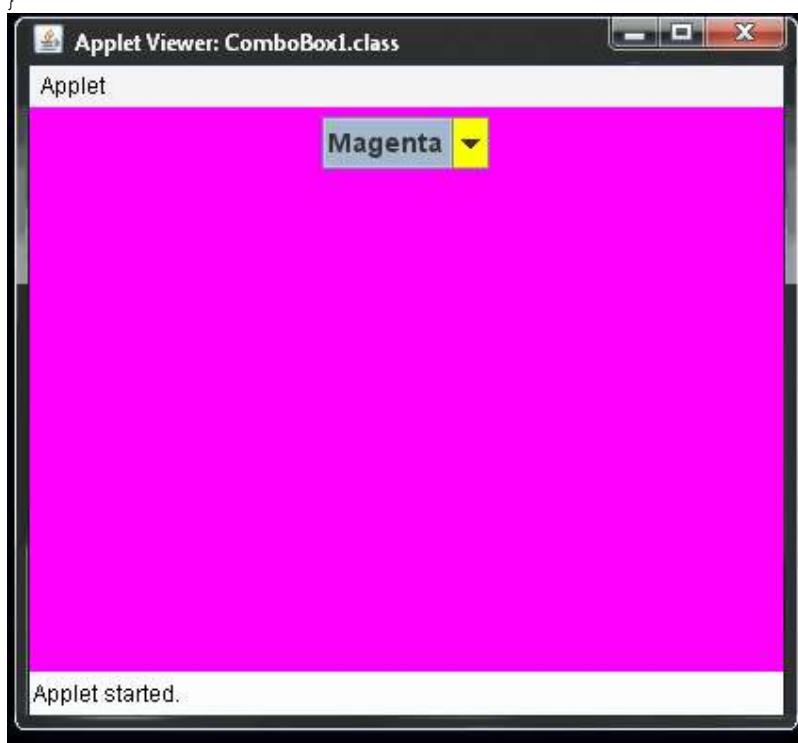
        /* Configuration */
        Selector.setForeground (Color.red);
        Selector.setFont (SansSerif);

        /* Decoration */
        Selector.setForeground (Color.red);
        Selector.setFont (SansSerif);

        /* Initialization */
        Selector.setSelectedIndex (5);
        Selector.setBackground (Color.yellow);
    }

    public void itemStateChanged(ItemEvent e) {
        int Selection;
        Selection = Selector.getSelectedIndex();
        if (Selection == 0) {
            Panel.setBackground (Color.red);
        } else if (Selection == 1) {
            Panel.setBackground (Color.magenta);
        } else if (Selection == 2) {
            Panel.setBackground (Color.blue);
        } else if (Selection == 3) {
            Panel.setBackground (Color.cyan);
        } else if (Selection == 4) {
    }
```

```
        Panel.setBackground (Color.green);
    } else if (Selection == 5) {
        Panel.setBackground (Color.yellow);
    } else if (Selection == 6) {
        Panel.setBackground (Color.white);
    } else if (Selection == 7) {
        Panel.setBackground (Color.gray);
    } else if (Selection == 8) {
        Panel.setBackground (Color.black);
    }
}
}
```



## JLabel

### Usage

The JLabel class can be used to add text (and icon) labels to a container.

### Package

javax.swing

### Class

javax.swing.JLabel

### Parent Classes

```
javax.swing.JLabel implements SwingConstants, Accessible  
javax.swing.JComponent implements Serializable  
java.awt.Container  
java.awt.Component implements ImageObserver, MenuContainer, Serializable  
java.lang.Object
```

## Common Public Constructors

### **JLabel ()**

Creates a blank JLabel instance.

### **JLabel (String Text)**

Creates a JLabel instance with the specified text.

### **JLabel (String Text, int Alignment)**

Creates a JLabel instance with the specified text and horizontal alignment.

## Common Public Methods

### **void setBackground (Color BgdColor)**

Sets the color of the background for the label.

### **void setHorizontalAlignment (int Alignment)**

Sets the alignment of the label's contents along the X axis.

### **void setFont (Font TextFont)**

Sets the font for this component.Arguments

### **void setForeground (Color TextColor)**

Sets the color of the text for the label.

### **void setText (String Text)**

Sets the text for the label.

## Arguments

### **Alignment**

The horizontal position of the text within the label. One of:

#### **JLabel.LEADING**

Identifies the leading edge of text for use with left-to-right and right-to-left languages.

#### **JLabel.CENTER**

The central position in an area.

#### **JLabel.TRAILING**

Identifies the trailing edge of text for use with left-to-right and right-to-left languages.

#### **JLabel.LEFT**

Box-orientation constant used to specify the left side of a box.

#### **JLabel.RIGHT**

Box-orientation constant used to specify the right side of a box.

### **BgdColor**

The color to be used for the background of the label.

### **Text**

The text to appear in the label.

### **TextColor**

The color to be used for the text of the label.

### **TextFont**

The font to be used for the text of the label.

## **JPasswordField Examples**

### **Usage:**

The JPasswordField class is a GUI component which can be used to add single line text input to a container, without displaying the entry.

### **Package:**

javax.swing

### **Subclass of:**

Java.awt.JTextField

### **Common Public Constructors:**

**JPasswordField ()**

Creates a blank JPasswordField instance.

**JPasswordField (String Text)**

Creates a JPasswordField instance with the specified text.

**JPasswordField (int Columns)**

Creates a blank JPasswordField instance with the specified number of columns.

**JPasswordField (String Text, int Columns)**

Creates a JPasswordField instance with the specified text and the specified number of columns.

### **Common Public Methods:**

**void addActionListener (ActionListener Handler)**

Configures an event handler for the PasswordField.

**char[] getPassword ()**

Returns the text in the field as an array of characters.

**void setBackground (Color BackgroundColor)**

Sets the background color of the PasswordField.

**void setText (String Text)**

Sets the text for the field.

### **Arguments:**

BackgroundColor

The color to be used for the background of the button.

Handler

The object which handles action events from this combo box.

Text

The text to be in the field.

### **Example:**

### **Code:**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PasswordField1 extends JApplet {
    /* Declaration */
    private JPasswordField Input;
```

```

private Container Panel;
private LayoutManager Layout;

public PasswordField1 () {
    /* Instantiation */
    Input = new JPasswordField ( 20);

    Layout = new FlowLayout ();
    Panel = getContentPane ();

    /* Location */
    Panel.setLayout (Layout);
    Panel.add (Input);

}

}

```



## JRadioButton Examples

### Usage:

The JRadioButton class can be used to add interactive grouped radio buttons to a container.

### Package:

`javax.swing`

### Subclass of:

`javax.swing.JToggleButton`

### Common Public Constructors:

`JRadioButton ()`

Creates a blank, unselected, JRadioButton instance.

`JRadioButton (String Text)`

Creates an unselected JRadioButton instance with the specified text.

`JRadioButton (String Text, boolean Selected)`

Creates a JRadioButton instance with the specified text and the designated selection state.

### Common Public Methods:

`void addItemListener (ItemListener Handler)`

Configures an event handler for the radio button.

`void setBackground (Color BackgroundColor)`

Sets the background color of the radio button.

**void setFont (Font TextFont)**

Sets the font for this component.

**void setForeground (Color TextColor)**

Sets the color of the text for the button.

**void setSelected (boolean State)**

Sets the state (checked or not checked) of the radio button.

**void setText (String Text)**

Sets the text for the button.

### **Arguments:**

BackgroundColor

The color to be used for the background of the button.

Handler

The object which handles action events from this combo box.

State

The checked state of the checkbox (true or false.)

Text

The text to appear in the button.

TextColor

The color to be used for the text of the button.

TextFont

The font to be used for the text of the button.

### **Example:**

#### **Code:**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioButton1 extends JApplet implements ItemListener {
    private Container Panel;
    private LayoutManager Layout;
    private JRadioButton Red;
    private JRadioButton Yellow;
    private JRadioButton White;
    private ButtonGroup Background;

    public RadioButton1 () {
        /* Instantiation */
        Layout = new FlowLayout ();
        Panel = getContentPane ();
        Red = new JRadioButton ("Red Background");
        Yellow = new JRadioButton ("Yellow Background", true);
        White = new JRadioButton ();
        Background = new ButtonGroup ();

        /* Location */
        Panel.setLayout (Layout);
        Panel.add (Red);
        Panel.add (Yellow);
        Panel.add (White);

        /* Decoration */
        Red.setBackground (Color.red);
```

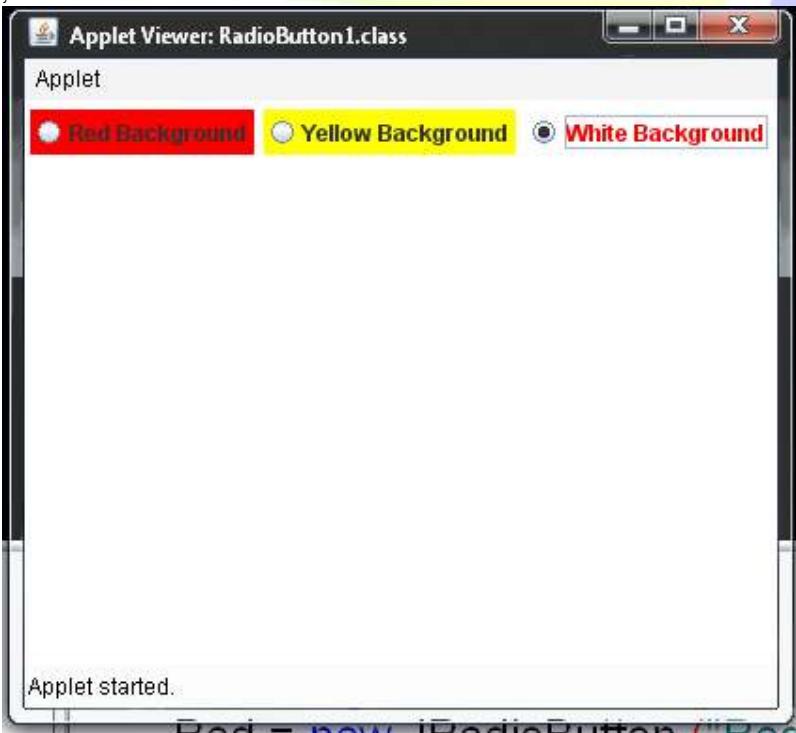
```

Yellow.setBackground (Color.yellow);
White.setBackground (Color.white);
White.setForeground (Color.red);
White.setText ("White Background");
Panel.setBackground (Color.yellow);

/* Configuration */
Background.add (Red);
Background.add (Yellow);
Background.add (White);
Red.addItemListener (this);
Yellow.addItemListener (this);
White.addItemListener (this);
}

public void itemStateChanged (ItemEvent e) {
    ItemSelectable Source;
    Source = e.getItemSelectable();
    if (Source == Red) {
        Panel.setBackground (Color.red);
    } else if (Source == Yellow) {
        Panel.setBackground (Color.yellow);
    } else if (Source == White) {
        Panel.setBackground (Color.white);
    }
}
}

```



## Java AWT Scrollbar Examples

V

### Usage:

The Scrollbar lets the user graphically select a value by sliding a knob within a bounded interval.

## **Package:**

Java.awt

## **Subclass of:**

Java.awt.Component

## **Common Public Constructors:**

**Scrollbar ()**

Creates a Scrollbar instance with a range of 0-100, an initial value of 0, and vertical orientation.

**Scrollbar (int Orientation)**

Creates a Scrollbar instance with a range of 0-100, an initial value of 0, and the specified orientation.

## **Common Public Methods:**

**void addAdjustmentListener (AdjustmentListener Handler)**

Configures an event handler for the scrollbar.

**int getValue ()**

Returns the value of the scrollbar setting.

**void setBackground (Color BackgroundColor)**

Sets the background color of the scrollbar.

**void setMaximum (int Max)**

Sets the maximum value of the scrollbar.

**void setMinimum (int Min)**

Sets the minimum value of the scrollbar.

**void setValue (int Value)**

Sets the current value of the scrollbar.

## **Arguments:**

BackgroundColor

The color to be used for the background of the field.

Handler

The object which handles action events from this field.

Max

The maximum value of the scrollbar.

Min

The minimum value of the scrollbar.

Orientation

The orientation of the scrollbar.

Scrollbar.HORIZONTAL

Orients the scrollbar horizontally.

Scrollbar.VERTICAL

Orients the scrollbar vertically.

Value

The current value of the scrollbar.

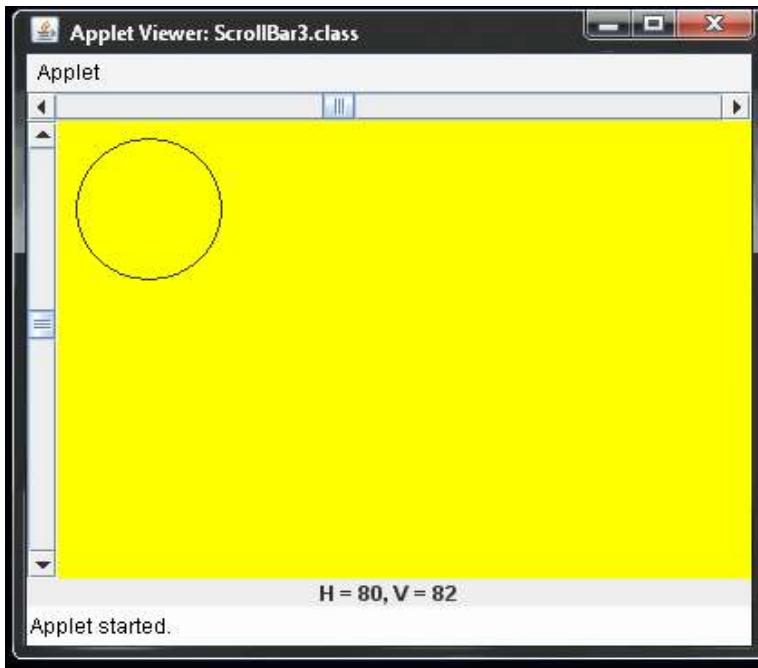
## **Example:**

### **Code:**

```
import java.awt.*;
import javax.swing.*;
import java.awt.event;
```

```
public class ScrollBar3 extends JApplet implements AdjustmentListener {  
    /* Declaration */  
    private Container Panel;  
    private LayoutManager Layout;  
    private JScrollBar HSelector;  
    private JScrollBar VSelector;  
    private Drawing Pad;  
    private JLabel Report;  
  
    public ScrollBar3 () {  
        /* Instantiation */  
        Layout = new BorderLayout ();  
        HSelector = new JScrollBar ();  
        VSelector = new JScrollBar ();  
        Report = new JLabel ();  
        Pad = new Drawing ();  
        Panel = getContentPane ();  
  
        /* Location */  
        Panel.setLayout (Layout);  
        Panel.add (Report, BorderLayout.SOUTH);  
        Panel.add (HSelector, BorderLayout.NORTH);  
        Panel.add (VSelector, "West");  
        Panel.add (Pad, "Center");  
  
        /* Configuration */  
        HSelector.setMaximum (200);  
        HSelector.addAdjustmentListener (this);  
        VSelector.setMaximum (200);  
        VSelector.addAdjustmentListener (this);  
        HSelector.setOrientation (JScrollBar.HORIZONTAL);  
  
        /* Decoration */  
        Report.setHorizontalAlignment (JTextField.CENTER);  
        Report.setText ("H = " + HSelector.getValue () +  
                       ", V = " + VSelector.getValue());  
        Pad.setBackground (Color.yellow);  
        Pad.setSize (220, 220);  
        Pad.setOval (HSelector.getValue (), VSelector.getValue ());  
    }  
  
    public void adjustmentValueChanged(AdjustmentEvent e) {  
        Report.setText ("H = " + HSelector.getValue () +  
                       ", V = " + VSelector.getValue ());  
        Pad.setOval (HSelector.getValue (), VSelector.getValue ());  
        Pad.repaint ();  
    }  
}  
  
class Drawing extends Canvas {  
    /* Declaration */  
    private int Width;  
    private int Height;  
  
    public void setOval (int X, int Y) {  
        Width = X;  
        Height = Y;  
    }  
    public void paint (Graphics g) {
```

```
    g.drawOval (10, 10, Width, Height);  
}  
}
```



## JSlider Examples

### Usage:

The JSlicer lets the user graphically select a value by sliding a knob within a bounded interval. Major tick marks and minor tick marks can be shown, both with variable periods.

### Package:

`javax.swing`

### Subclass of:

`javax.swing.JComponent`

### Common Constructors:

`JSlider ()`

Creates a JSlider instance with a range of 0-100, an initial value of 0, and horizontal orientation.

`JSlider (int Orientation)`

Creates a JSlider instance with a range of 0-100, an initial value of 0, and the specified orientation.

`JSlider (int Min, int Max)`

Creates a JSlider instance with a range of Min-Max, an initial value of  $(\text{Min} + \text{Max})/2$ , and horizontal orientation.

`JSlider (int Orientation, int Min, int Max)`

Creates a JSlider instance with a range of Min-Max, an initial value of  $(\text{Min} + \text{Max})/2$ , and the specified orientation.

`JSlider (int Orientation, int Min, int Max, int Value)`

Creates a JSlider instance with a range of Min-Max, an initial value as specified, and the specified orientation.

## Common Methods:

```
addChangeListener (ChangeListener Handler)
    Configures an event handler for the Slider.
getValue ()
    Returns the value of the slider setting.
setBackground (Color BackgroundColor)
    Sets the background color of the Slider.
setInverted (boolean Inverted)
    Reverses the sense of Min and Max iff true.
setMajorTickSpacing (int Delta)
    Set the spacing of the major tick marks.
setMaximum (int Max)
    Sets the maximum value of the slider.
setMinorTickSpacing (int Delta)
    Set the spacing of the minor tick marks.
setMinimum (int Min)
    Sets the minimum value of the slider.
setValue (int Value)
    Sets the current value of the slider.
```

## Arguments:

BackgroundColor  
The color to be used for the background of the field.

Delta  
The spacing of the tick marks.

Handler  
The object which handles action events from this field.

Max  
The maximum value of the slider.

Min  
The minimum value of the slider.

Orientation  
The orientation of the slider.  
JSlider.HORIZONTAL  
Orients the slider horizontally.  
JSlider.VERTICAL  
Orients the slider vertically.

Value  
The current value of the slider.

## Example:

### Code:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
public class Slider2 extends JApplet implements ChangeListener {
    private Container Panel;
    private LayoutManager Layout;
    private JSilder HSelector;
    private JSilder VSelector;
    private Drawing Pad;
    private JLabel Report;

    public Slider2 () {
        Layout = new BorderLayout ();
        HSelector = new JSilder ();
        VSelector = new JSilder (JSilder.VERTICAL, 0, 200, 100);
        Report = new JLabel ();
        Pad = new Drawing ();
        Panel = getContentPane ();

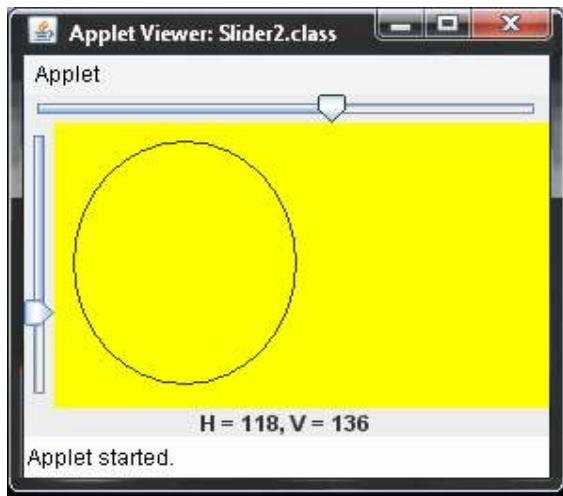
        HSelector.setMaximum (200);
        HSelector.addChangeListener (this);
        VSelector.setInverted (true);
        VSelector.addChangeListener (this);
        Panel.setLayout (Layout);
        Panel.add (Report, BorderLayout.SOUTH);
        Panel.add (HSelector, BorderLayout.NORTH);
        Panel.add (VSelector, "West");
        Panel.add (Pad, "Center");
        Pad.setBackground (Color.yellow);
        Pad.setSize (220, 220);
        Pad.setOval (HSelector.getValue (), VSelector.getValue ());
        Report.setHorizontalAlignment (JTextField.CENTER);
        Report.setText ("H = " + HSelector.getValue () +
                        ", V = " + VSelector.getValue ());
    }

    public void stateChanged (ChangeEvent e) {
        Report.setText ("H = " + HSelector.getValue () +
                        ", V = " + VSelector.getValue ());
        Pad.setOval (HSelector.getValue (), VSelector.getValue ());
        Pad.repaint ();
    }
}

class Drawing extends Canvas {
    private int Width;
    private int Height;

    public void setOval (int X, int Y) {
        Width = X;
        Height = Y;
    }

    public void paint (Graphics g) {
        g.drawOval (10, 10, Width, Height);
    }
}
```



## JTextArea Examples

### Usage:

The JTextArea class can be used to multiple line text output to a container.

### Package:

`javax.swing`

### Subclass of:

`javax.swing.JTextComponent`

### Common Constructors:

`JTextArea ()`

Creates a blank JTextArea instance.

`JTextArea (String Text)`

Creates a JTextArea instance with the specified text.

`JTextArea (int Rows, int Columns)`

Creates a blank JTextArea instance with the specified number of columns.

`JTextArea (String Text, int Rows, int Columns)`

Creates a JTextArea instance with the specified text and the specified number of columns.

### Common Methods:

`getText ()`

Returns the text in the field.

`setBackground (Color BackgroundColor)`

Sets the background color of the TextArea.

`setEditable (boolean Editable)`

Sets the field as being editable or fixed.

`setFont (Font TextFont)`

Sets the font for this component.

`setLineWrap (boolean LineWrap)`

Sets the font for this component.

`setText (String Text)`

Sets the text for the field.

### Arguments:

BackgroundColor

The color to be used for the background of the area.  
Columns  
The width of the text area.  
Editable  
The state of the area as editable or fixed.  
Handler  
The object which handles action events from this combo box.  
LineWrap  
True for line wrap, false for no wrap.  
Rows  
The height of the text area.  
Text  
The text to appear in the area.  
TextFont  
The font to be used for the text of the button.

## JTextField Examples

### Usage:

The JTextField class is a GUI component which can be used to add single line text input to a container.

### Package:

Java.awt.swing

### Subclass of:

Java.awt.JTextField

### Common Public Constructors:

`JTextField ()`

Creates a blank JTextField instance.

`JTextField (String Text)`

Creates a JTextField instance with the specified text.

`JTextField (int Columns)`

Creates a blank JTextField instance with the specified number of columns.

`JTextField (String Text, int Columns)`

Creates a JTextField instance with the specified text and the specified number of columns.

### Common Public Methods:

`void addActionListener (ActionListener Handler)`

Configures an event handler for the TextField.

`String getText ()`

Returns the text in the field.

`void setBackground (Color BackgroundColor)`

Sets the background color of the TextField.

`void setEditable (boolean Editable)`

Sets the field as being editable or fixed.

`void setFont (Font TextFont)`

Sets the font for this component.

`void setText (String Text)`

Sets the text for the field.

## **Arguments:**

BackgroundColor

The color to be used for the background of the field.

Editable

The state of the field as editable or fixed.

Handler

The object which handles action events from this field.

Text

The text to appear in the field.

TextFont

The font to be used for the text of the button.

## **BoxLayout**

### **Usage**

The BoxLayout class is a Layout Manager which can be used to arrange controls in a container. BoxLayouts can be either vertical or horizontal.

### **Package**

Javax.swing

### **Subclass of**

java.lang.Object

java.awt.FlowLayout implements LayoutManager2, Serializable

### **Public Constructors**

**BoxLayout (Container Target, int Orientation)**

Creates a blank BoxLayout instance.

### **Public Fields**

#### **X\_AXIS**

Specifies that components should be laid out left to right.

#### **Y\_AXIS**

Specifies that components should be laid out top to bottom.

### **Example:**

```
import java.applet.Applet;
import java.awt.*;
```

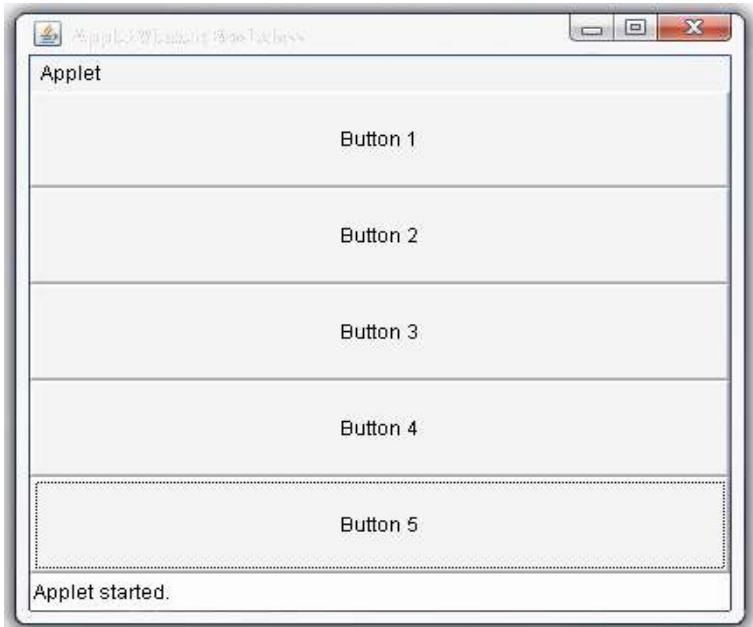
### **Code:**

```
import javax.swing.*;  
  
public class Box1 extends Applet {  
    LayoutManager Layout;  
    Button [] Buttons;  
  
    public Box1 () {  
        int i;  
  
        Layout = new BoxLayout (this, BoxLayout.X_AXIS);  
        setLayout (Layout);  
  
        Buttons = new Button [5];  
        for (i = 0; i < 5; ++i) {  
            Buttons[i] = new Button ();  
            Buttons[i].setLabel ("Button " + (i + 1));  
            add (Buttons[i]);  
        }  
    }  
}
```

## Example

## Code

```
import java.applet.Applet;  
import java.awt.*;  
import javax.swing.*;  
  
public class Box2 extends Applet {  
    LayoutManager Layout;  
    Button [] Buttons;  
  
    public Box2 () {  
        int i;  
  
        Layout = new BoxLayout (this, BoxLayout.Y_AXIS);  
        setLayout (Layout);  
  
        Buttons = new Button [5];  
        for (i = 0; i < 5; ++i) {  
            Buttons[i] = new Button ();  
            Buttons[i].setLabel ("Button " + (i + 1));  
            add (Buttons[i]);  
        }  
    }  
}
```



## java.awt.GridBagLayout

### Description

**GridBagLayout** lays out components based on a grid with rows and columns that need not all be the same size. If you're familiar with HTML tables, you'll feel comfortable with the idea. Unfortunately, **GridBagLayout** is infernally awkward and error-prone to use. After reading some of this you might want to see example code at [Example: GridBagLayout](#).

### Abandon hope, all ye who use GridBagLayout

Altho **GridBagLayout** can produce acceptable results, it's not a happy story.

- It is very difficult to work with because eleven *constraint values* are used for each component! These constraint values are bundled in a `java.awt.GridBagConstraints` object. Fortunately, most of them have reasonable defaults.
- The layout doesn't have features which specifically help to use Sun's (or anyone else's) *Human Interface Guidelines*, for example in handling gaps.
- Spacing is specified in pixels, which provides no flexibility for screen resolution and font changes.
- I like this quote from Otaku, Cedric's blog: "GridBagLayout is an absolute disaster and the perfect example of something that is completely flawed and violates with a stunning regularity the principle of least surprise."
- Everyone who has touched **GridBagLayout** will enjoy *Totally Gridbag*, little bit of Flash foolishness - [madbean.com/blog/2004/17/totallygridbag.html](http://madbean.com/blog/2004/17/totallygridbag.html)
- Some prefer to call it "GridBugLayout".

**Subpanels for unrelated components.** An entire window may require use of nested panels -- don't try to force everything into one giant **GridBagLayout**. If some groups of components (eg, radio buttons, groups of buttons, checkboxes) are unrelated to alignment with other components, put them in their own panel and use an appropriate layout (often **GridLayout**).

### Unequal rows and columns

The underlying idea is *rows and columns*, similar to HTML tables.

- Rows and columns don't all have to be same size. The size of each row and column is adjusted depending on maximum component size in the row/column and their constraints.

- A component *display area* may span several rows and/or columns. Describe this area by giving the row, column, width (in columns), and height (in rows).
- A component doesn't have to fill its display area; constraints describe how to align it within its area.

### Three categories of constraint parameters

GridLayout performs three functions using values from the GridBagConstraints parameter in the add() method. The names [note the case violations of the standard Java naming convention] are the following.

1. Grid position, width and height describe the display area.. gridx, gridy, gridwidth, and gridheight.
2. Position within the display area. fill, anchor, insets, ipadx, and ipady.
3. Identifying rows and columns which receive extra space on expansion. weightx, and weighty.

### Constructor

```
JPanel p = new JPanel(new GridLayout());
or
```

```
JPanel p = new JPanel();
p.setLayout(new GridLayout());
```

Do not give the number of rows and columns when you create a new GridLayout. The are computed by the layout manager as a consequence of what you add.

### Adding a component

Components are added to JPanel p like this:

```
p.add(component, constraintObj);
```

Constraint parameters are instance fields in a GridBagConstraints object. The add() method clones (copies) the constraint object, so typically the same GridBagConstraints object is reused to pass constraints for the next component. It is common to use only one GridBagConstraints object to add all components, changing fields as necessary between adds, but this practice also leads to many errors and code which is difficult to change.

### GridBagConstraints fields

The fields can divided into three groups:

Necessary - row and column coordinates, width and height in terms of rows and columns

Describe the location and size (in rows and columns) of the component using gridx, gridy, gridwidth, and gridheight.

Common - expandability and alignment

If a column or row can use extra space when a window is expanded, set weightx and weighty to 1.0; you will rarely want to use other values.

Use fill depending on what the type of component is. For example, a JTextField can usually grow horizontally, so you would use  
`gbc.fill = GridBagConstraints.HORIZONTAL;`  
to allow it to stretch.

Use anchor to tell which edges of the display area a component should be attached to.

Less common - surrounding spacing

You can also control the space around a component with ipadx and ipady.

To control the amount of unused space around a component, use insets.

### GridBagConstraints attribute summary

|              |                                                                                                                                                                                                                                                     |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gridx</b> | The int column (gridx) and row (gridy) of the component. If requires more than one cell (gridwidth or gridheight > 1), this is the coordinate of the top-left cell. The row and columns start at zero. The value GridBagConstraints.RELATIVE places |
| <b>gridy</b> |                                                                                                                                                                                                                                                     |

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | <p>the component in the next position.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>gridwidth</b>  | Number of columns or rows the component occupies.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>gridheight</b> | GridBagConstraints.REMAINDER indicates that this component should fill out all rows or columns to the end. Default 1. Note these names violates naming standards (second word not capitalized).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>weightx</b>    | These double variables (default value 0) are used in calculating where space is allocated in rows and columns when a window is resized. extra space should be allocated in a column horizontally ( <b>weightx</b> ) and row vertically ( <b>weighty</b> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>weighty</b>    | A column width is originally calculated as the maximum width of the preferred sizes of the components in that column. If a component is narrower than the column width, the value of the <b>fill</b> attribute is used in deciding how to use that extra space (see below).<br>Extra horizontal space is allocated to a column in proportion to the maximum weightx value for that column. If weightx is zero, no extra space is allocated. Because these weights are <i>relative</i> , you can assign any arbitrary positive double value to produce the desired layout, altho it is common to work in the range 0-1. The analogous procedure is performed for the row heights using <b>weighty</b> .                                                                                                  |
| <b>fill</b>       | <b>Fill</b> specifies how the component should expand within its display area if the area width/height is larger than its preferred size.<br><pre>GridBagConstraints.NONE      // Can't expand (Default) GridBagConstraints.VERTICAL // Expand vertically GridBagConstraints.HORIZONTAL // Expand horizontally GridBagConstraints.BOTH     // Expand vertically and horizontally</pre> For example, a text field typically expands horizontally (GridBagConstraints.HORIZONTAL), a text area expands vertically and horizontally (GridBagConstraints.BOTH), and a button might not expand at all (GridBagConstraints.NONE). If a component isn't expanded to fill the space, the anchor attribute (see below) is used to specify where it goes within that cell. Default value GridBagConstraints.NONE. |
| <b>anchor</b>     | If the component doesn't occupy its entire display area, <b>anchor</b> specifies where it should be placed. The location is usually given as a compass direction. (A relative system which works for both right-to-left and left-to-right languages is also available).<br><pre>GridBagConstraints.CENTER (the default), GridBagConstraints.NORTH   GridBagConstraints.SOUTH GridBagConstraints.NORTHEAST GridBagConstraints.SOUTHWEST GridBagConstraints.EAST    GridBagConstraints.WEST GridBagConstraints.SOUTHEAST GridBagConstraints.NORTHWEST</pre>                                                                                                                                                                                                                                               |
| <b>insets</b>     | A java.awt.Insets object adds padding space to the component. Insets should be rarely used because they often produce bad results, with the exception of JPanels and JLabels. The constructor parameters (in pixels) specify the top, left, bottom, right. For example,<br><pre>gbc.insets = new Insets(10, 5, 10, 4);</pre> Default value: Insets(0,0,0,0).                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>ipadx</b>      | These int fields specify an increase or decrease in the horizontal and/or vertical preferred size of the component. Default value 0. Negative values can be used to tighten the spacing. These values are rarely useful, altho they can be used for fine-tuning spacing.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>ipady</b>      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

## Java Idiom

It's common to write a utility method which sets the fields of a GridBagConstraints object.

For example,

```
GridBagConstraints gbc = new GridBagConstraints();
. . .
private void set_gbc(int row, int column, int width, int height, int fill)
{
    gbc.gridx = row;
    gbc.gridy = column;
    gbc.gridwidth = width;
    gbc.gridheight = height;
    gbc.fill = fill; // GridBagConstraints.NONE .HORIZONTAL .VERTICAL .BOTH
    // leave other fields (eg, anchor) unchanged.
}
. . .
set_gbc(3, 2, 2, 1, GridBagConstraints.HORIZONTAL);
gbc.insets = new Insets(3,3,3,3); // put spacing around this field.
p.add(myTextField, gbc);
```

## Adding a component

To add myField to the third row (2) and first column (0), where the button will be one grid cell wide and one high, and it will be able to expand horizontally, do this:

```
set_gbc(2, 0, 1, 1, GridBagConstraints.HORIZONTAL);

gridbag.setConstraints(myField, gbc);
panel.add(myField);
```



# Java Networking (Sockets)

## **Computer Networking**

Here we are going to unfold the concepts of networking in day to day scenarios. Let us consider that you started a commercial activity where you were using a computer along with a printer, modem and a CD-ROM. Very soon you need to expand your business and you felt the requirement to add more people as well more computers. Suppose you buy 5 computers instead of going for a decision to buy 5 more printers, you should opt for an environment having the capabilities to share the available printer by the new 5 computers, this will save the money and resources to buy and manage the 5 new printers. Such an environment

providing the sharing and exchanging facilities among the computer machines is called as a computer network.



Now you just need to maintain a network in which all the computers are interconnected and share the same printer and all such devices i.e. we just link the computer devices together with hardware and software supporting data communication across the network.



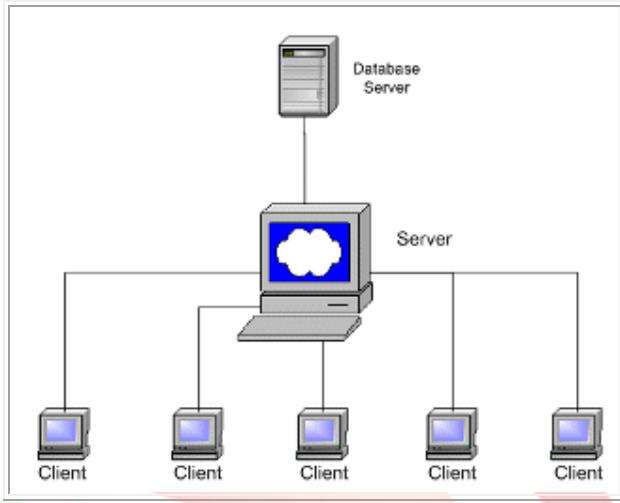
The initial and primary justification for a network is to avoid the multiple iterations of documents, files and spreadsheets on disparate computers.

### **Client-Server Architecture**

*Client-server architecture can be considered as a network environment that exchanges information between a server machine and a client machine where server has some resources that can be shared by different clients.*

*In a Client/server architecture individual computers (known as clients) are connected to a central computer which is known as “server”.*

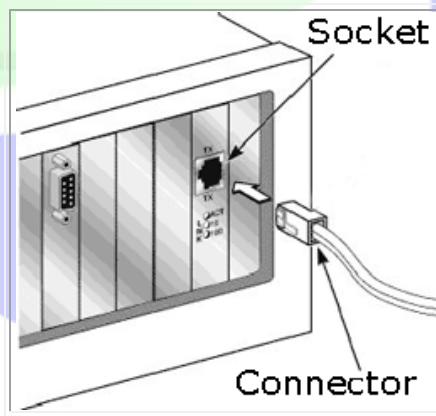
Let's take an example of a file server to understand the core process of a client/server network, the file server acts as a storage space on the network for the files, spreadsheets, databases, etc. Instead of storing these records on every individual computer, the file server allows the clients to store their files on one central computer and make them sharable. The client-server architecture is beneficial in reducing the multiple iterations of a single file and allowing the organization to have one centralized point for every computer to access the same file.



The interaction between a lamp and an electrical socket can be considered as a interaction between client and server is just like. In the example the electrical socket is just like a server and the lamp works like a client.

## Socket and ports

In common language we can say that the socket is one of the most primal technologies of computer networking. *Sockets are just like an end-point of two-way communication, which allow applications to communicate using network hardware and operating systems.* However in case of java never get confused with a socket. *Socket classes are used to establish a connection between client program and a server program. In java there is a java.net package, which provides two types of classes- first is ordinary socket, which implement the client side connection and second is server socket, which implement the server side connection.*



*The main purpose of the server socket is to listen an incoming connection request and ordinary socket is used to ask to server for the connection. Once a connection between client and server established, both the connected socket can communicate with each other.*

## Introduction of networking Ports

*In other hand we can consider the work of port in connection-based communication is like a medium through which, an application establish a connection with another application by binding a socket by a port number. Addressing the information and the port no., accompanied the data transfer over the network. The Ports are used by TCP and UDP to deliver the data to the right application, are identified by a 16-bit number.*

*There is a limitation for the port that no port can be bound by two applications at the same time.*

**Example:** If we consider a letter (data packet) sent to a particular apartment (IP) with house no. (port no), at this time the port no. is the most important part for the delivery of the letter. In order for the delivery to work, the sender needs to include an house number along with the address to ensure the letter gets to the right destination.

If we consider the client-server architecture, a server application binds a socket to a specific port number in connection-based communication. It registered the server with the system where all the data destined for that port.

Now we are aware of the importance of the port number. In the same order there are some ports which are predefine and called reserved ports. Some of them are given below :-

**Reserved port numbers.**

| Service | Port no. |
|---------|----------|
| echo    | 7        |
| daytime | 13       |
| ftp     | 21       |
| telnet  | 23       |
| Smtp    | 25       |
| Finger  | 79       |
| http    | 80       |
| pop3    | 110      |

If we consider the range of the port numbers, there are 0 to 65,535 ports available. The port numbers ranging from 0 - 1023 are reserved ports or we can say that are restricted ports. All the 0 to 1023 ports are reserved for use by well-known services such as FTP, telnet and http and other system services. These ports are called well-known ports.

## **URL in term of Java Network Programming**

A URL (Uniform Resource Locator) is the address of a resource on the Internet. In java network programming we can use URLs to connect and retrieve information over the Internet. In this section we will provide the complete information about the way of using URL in java network programming through a full code-example. The example give you the full exposure to network programming e.g. how to open a connection to a URL, how to retrieve information of a given URL.

Now here we are provide a simple name structure of a URL which addresses the Java Web site hosted by roseindia:

`http://java.sun.com`

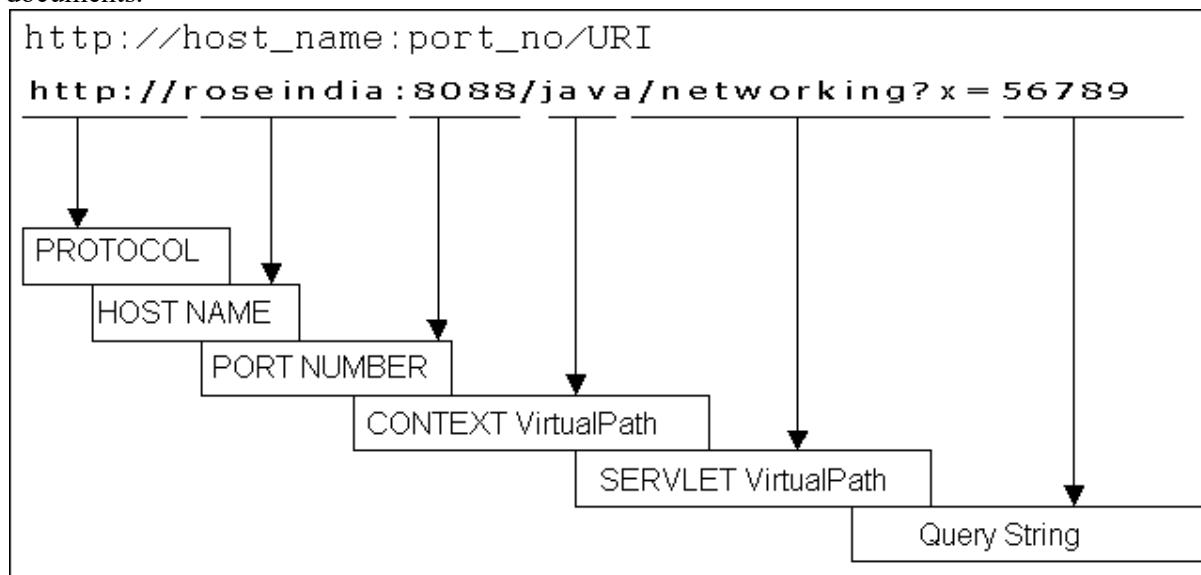
Protocol Identifier

Resource Name

In the above given URL structure there are two main components:

- Protocol identifier
- Resource name

In which there is a colon and two forward slashes between protocol identifier and the resource name. The protocol identifier is the name of the protocol which is used in the URL structure to get the resource. Here we use the http which is one of the protocols used to access different types of resources on the net. HTTP is mainly used to serve the hypertext documents.



The second part of the URL is resource name that is the complete address of the resource. The resource name contains one or more of the components listed in the following table :-

|                    |                                                                                                                                 |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <b>Host Name</b>   | The name of the machine on which the resource lives.                                                                            |
| <b>Filename</b>    | The pathname to the file on the machine.                                                                                        |
| <b>Port Number</b> | The port number to which to connect (typically optional).                                                                       |
| <b>Reference</b>   | A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional). |

In the above given table of the URL structure, for many protocols the host name and the filename are required but in some times the port number and reference are optional.

## NetWorking in Java

In this section we are exploring the **java.net** package which provides the support for networking in java with a generic style. All the java classes for developing a network program are defined in the **java.net** package.

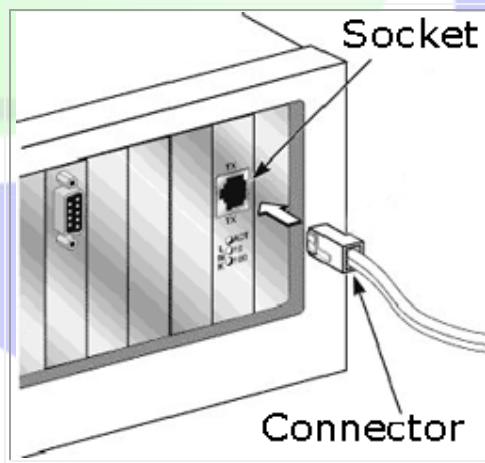
All the java networking classes and interfaces are contained in the `java.net` package, it is given below:

|                                 |                                     |                               |
|---------------------------------|-------------------------------------|-------------------------------|
| <code>Authenticator</code>      | <code>JarURLConnection</code>       | <code>SocketPermission</code> |
| <code>ContentHandler</code>     | <code>MulticastSocket</code>        | <code>URL</code>              |
| <code>DatagramPacket</code>     | <code>NetPermission</code>          | <code>URLClassLoader</code>   |
| <code>DatagramSocket</code>     | <code>PasswordAuthentication</code> | <code>URLConnection</code>    |
| <code>DatagramSocketImpl</code> | <code>ServerSocket</code>           | <code>URLDecoder</code>       |
| <code>HttpURLConnection</code>  | <code>Socket</code>                 | <code>URLEncoder</code>       |
| <code>InetAddress</code>        | <code>SocketImpl</code>             | <code>URLStreamHandler</code> |

## **Server Sockets**

In common language we can say that the sockets are just like an end-point of two-way communication link over the network between two programs. Socket classes are used to establish a connection between client program and a server program. In java there is a `java.net` package, which provides two types of classes- first is ordinary socket, which implement the client side connection and second is server socket, which implement the server side connection.

*In Java there are many socket class that is used for creating a Server applications. ServerSockets are quite different from normal Sockets. The main work of `ServerSocket` class is to wait for a request of connection by the client and connect them on published ports and then possibly returns a result to the requester. The `SocketImpl` is a common superclass of all classes that actually implement sockets. It is used to create both client and server sockets.*



There are some constructors that might throw an `IOException` under adverse conditions. Some of the constructors are as under:

|                                                                             |                                                                                                                      |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>ServerSocket(int port)</code>                                         | Creates server socket on the specified port with a queue length of 50.                                               |
| <code>ServerSocket(int port, int maxQueue)</code>                           | Creates a server socket on the specified port with a maximum queue length of <code>maxQueue</code> .                 |
| <code>ServerSocket(int port, int maxQueue, InetAddress localAddress)</code> | Creates a server socket on the specified port with a maximum queue length of <code>maxQueue</code> . On a multihomed |

host, `localAddress` specifies the IP address to which this socket binds.

### **Methods Of Server Socket**

The `accept()` and `close()` methods provide the basic functionality of a server socket.

```
public Socket accept() throws IOException  
public void close() throws IOException
```

On a server with multiple IP addresses, the `getInetAddress()` method tells you which one this server socket is listening to. The `getLocalPort()` method tells you which port you're listening to.

```
public InetAddress getInetAddress()  
public int getLocalPort()
```

# Sockets

Before data is sent across the Internet from one host to another using TCP/IP, it is split into packets of varying but finite size called *datagrams*. Datagrams range in size from a few dozen bytes to about 60,000 bytes. Anything larger than this, and often things smaller than this, needs to be split into smaller pieces before it can be transmitted. The advantage is that if one packet is lost, it can be retransmitted without requiring redelivery of all other packets. Furthermore if packets arrive out of order they can be reordered at the receiving end of the connection.

However this is all transparent to the Java programmer. The host's native networking software transparently handles the splitting of data into packets on the sending end of a connection, and the reassembly of packets on the receiving end. Instead, the Java programmer is presented with a higher level abstraction called a *socket*. The socket represents a reliable connection for the transmission of data between two hosts. It isolates you from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.

***There are four fundamental operations a socket performs. These are:***

1. *Connect to a remote machine*
2. *Send data*
3. *Receive data*
4. *Close the connection*

A socket may not be connected to more than one host at a time.

The `java.net.Socket` class allows you to perform all four fundamental socket operations. You can connect to remote machines; you can send data; you can receive data; you can close the connection.

Connection is accomplished through the constructors. Each `Socket` object is associated with exactly one remote host. To connect to a different host, you must create a new `Socket` object.

### ***Constructors***

#### **1) `Socket`**

```
public Socket(String host,  
             int port) throws UnknownHostException, IOException
```

Creates a stream socket and connects it to the specified port on the specified host.

#### **Parameters:**

host - the host

port - the port

#### **2) `Socket`**

```
public Socket(String host,  
             int port,  
             boolean stream) throws IOException
```

Creates a socket and connects it to the specified port on the specified host. The last argument lets you specify whether you want a stream or datagram socket.

**Parameters:**

host - the specified host

port - the specified port

stream - a boolean indicating whether this is a stream or datagram socket

**3) Socket**

```
public Socket(InetAddress address,  
             int port) throws IOException
```

Creates a stream socket and connects it to the specified address on the specified port.

**Parameters:**

address - the specified address

port - the specified port

**4) Socket**

```
public Socket(InetAddress address,  
             int port,  
             boolean stream) throws IOException
```

Creates a socket and connects it to the specified address on the specified port. The last argument lets you specify whether you want a stream or datagram socket.

**Parameters:**

address - the specified address

port - the specified port

stream - a boolean indicating whether this is a stream or datagram socket

**Methods Of Sockets**

**1) getInetAddress**

```
public InetAddress getInetAddress()  
Gets the address to which the socket is connected.
```

**2) getPort**

```
public int getPort()  
Gets the remote port to which the socket is connected.
```

**3) getLocalPort**

```
public int getLocalPort()  
Gets the local port to which the socket is connected.
```

**4) getInputStream**

```
public InputStream getInputStream() throws IOException  
Gets an InputStream for this socket.
```

**5) getOutputStream**

```
public OutputStream getOutputStream() throws IOException  
Gets an OutputStream for this socket.
```

**6) close**

```
public synchronized void close() throws IOException  
Closes the socket.
```

## **Client Server Chat Program Using Socket/ServerSocket**

**Program:server.java**

//Server program

```
import java.io.*;
```

```

import java.net.*;
public class servers implements Runnable
{ ServerSocket server;
PrintStream stoc;
DataInputStream fromc;
Socket client;
Thread thread;
public servers()
{ try{ server=new ServerSocket(1001);//open port
}catch(Exception e)
{ System.out.println("Fail to create socket");}
thread=new Thread(this);
thread.start();
}
public void run()
{ try{ while(true)
{ client=server.accept(); //wait for client
//create Stream which read data from client buffer
fromc =new DataInputStream((client.getInputStream()));
//read chars form client buffer
String str=fromc.readLine();
System.out.println("Client says:"+str);
//create stream to write information in client buffer
stoc=new PrintStream(client.getOutputStream());
//read data from keyboard
DataInputStream X=new DataInputStream(System.in);
System.out.print("Server Message:");
String msg=X.readLine();
//send msg to client
stoc.println(msg);
}}catch(Exception e){ System.out.println(e);}
try{ client.close();
}catch(Exception e)
{ System.out.println("Fail to close client connection");} }
}

public static void main(String arg[])
{ new servers();}

```

**Program: clients.java**

```

//client program
import java.io.*;
import java.net.*;
public class clients
{

```

```

PrintStream ctos;
DataInputStream froms;

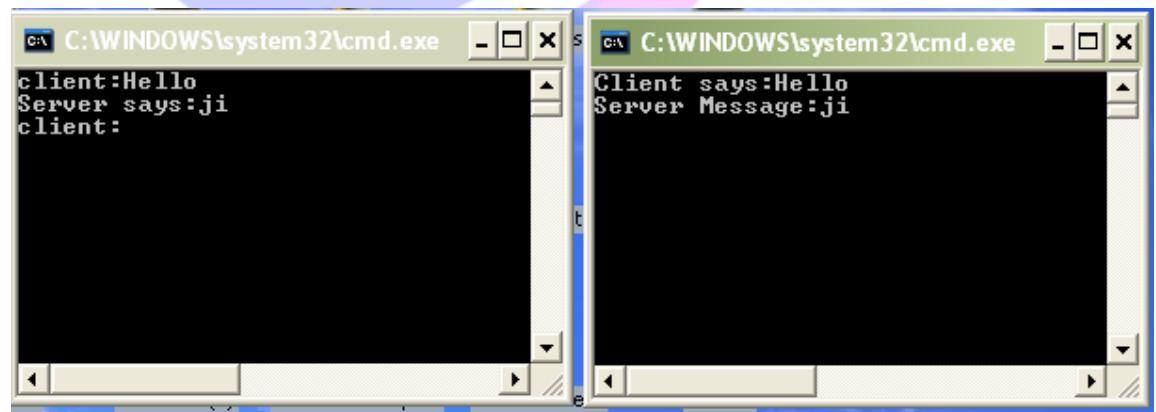
Socket toserver;

public clients()
{
    connectToServer();
}

private void connectToServer()
{ try{
    while(true)
    {toserver=new Socket("localhost",1001);
    //create stream to read data from stream send by server
    froms=new DataInputStream((toserver.getInputStream()));
    DataInputStream X=new DataInputStream(System.in);
    System.out.print("client:");
    String msg=X.readLine();
    //create stream to write data
    ctos=new PrintStream(toserver.getOutputStream());
    //write data in stream
    ctos.println(msg);
    //read data from stream send by server
    String str=froms.readLine();
    System.out.println("Server says:"+str);
}
    }catch(Exception e){ System.out.println(e);}
}

public static void main(String arg[])
{ new clients();
}

```



## Obtaining Internet address information

### Class InetAddress

*This class provides methods to access host names and IP addresses.*

The following methods are provided to create InetAddress objects:

**Static InetAddress getLocalHost() throws UnknownHostException** This method returns an InetAddress object for the local machine.

**Static InetAddress getByName(String host) throws UnknownHostException** This method returns an InetAddress object for the specified host name. The host name can be either a mnemonic identifier such as "www.Java.com" or an IP address such as 121.1.28.54. This is the only method that can be used by a client to get remote hosts details.

The following methods are provided to extract information from and InetAddress object.

**byte[] getAddress()** This method returns an array of bytes corresponding to the IP address held in the InetAddress object. The array is in network byte order i.e. high byte first (back to front) and must be converted to a string before being displayed to the screen.

**String getHostName()** Returns the host name held in the InetAddress object. If the host name is not already known an attempt is made to look it up, if this fails the IP address is returned as a string.

#### *Exception Errors*

**UnknownHostException** This is a subclass of IOException and indicates that the host name could not successfully be identified.

**SecurityException** This error is thrown if the Java security manager has restricted the desired action from taking place. Currently an Applet may only create an InetAddress object for host that it originated from, i.e. the host web server. Any attempt to create an InetAddress object for another host will throw this exception.

### Class InetAddress sample code

The method below, getLocalnetdetails(), simply creates an InetAddress object and retrieves the local host name and IP address. It uses another method called toText to convert the byte array IP address to a string so that it can be sent to the TextArea object called "output".

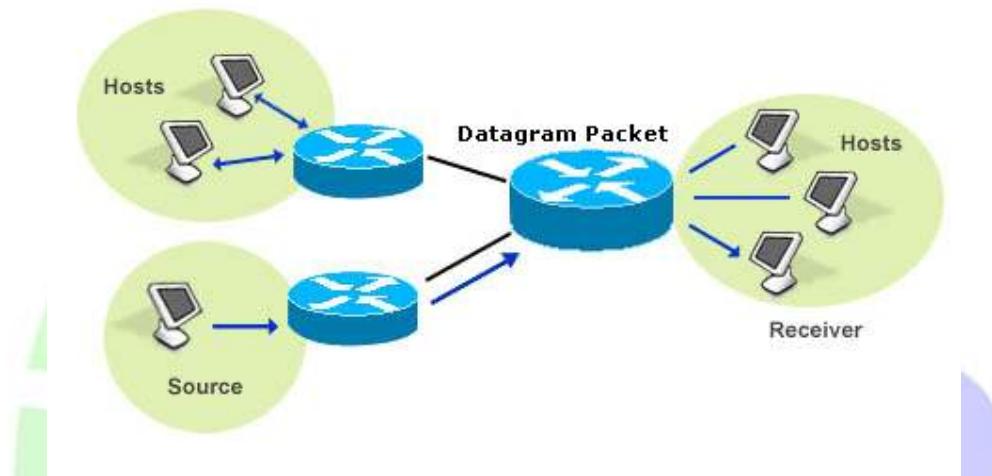
```
import java.net.*;
class cc
{
public static void main(String arg[])
{
try {
    InetAddress myself = InetAddress.getLocalHost ();
    System.out.println("Local hostname : " + myself.getHostName () + "\n");
    byte b[]={myself.getAddress()};
    String ip="";
    for(int i=0;i<b.length;i++)
    {ip=ip+b[i]+".";}

    System.out.println("Local IP Address : " + ip + "\n");
}
catch (UnknownHostException ex){
    System.out.println("Failed to find myself:");
}
}
}
```

## **Datagram in network environment**

In a network environment the client and the server communicate with each-other by reliable channel like TCP socket which have dedicated point-to-point channel between client and server. All data sent over the channel is received and sent in the same order.

In other case the application, which communicate by datagrams sends and receives completely independent packets of information and they also don't need a dedicated point-to-point channel.



Datagrams are simply a bundles of information data passed between machines. Java implements datagrams on top of the UDP protocol by using three classes which are in java.net package as well as define as under :

- DatagramSocket
- DatagramPacket
- MulticastSocket

In which the DatagramPacket is used to contain data for sent and receive by a application and the DatagramSocket is used to send or receive the DatagramPackets over the network envirnoment. While the Multicast socket is used to broadcast the DatagramPackets to multiple recipients.

## **Introducing UDP**

The *User Datagram Protocol*, UDP for short, provides unguaranteed, connectionless transmission of data across an IP network. By contrast, TCP, provides reliable, connection-oriented transmission of data.

Both TCP and UDP split data into packets called *datagrams*. However TCP includes extra headers in the datagram to enable retransmission of lost packets and reassembly of packets into the correct order if they arrive out of order. UDP does not provide this. If a UDP packet is lost, it's lost. It will not be retransmitted. Similarly, packets appear in the receiving program in the order they were received, not necessarily in the order they were sent.

Given these disadvantages you may well wonder why anyone would prefer UDP to TCP. The answer is speed. UDP can be up to three times faster than TCP; and there are many applications for which reliable transmission of data is not nearly as important as speed. For

example lost or out of order packets may appear as static in an audio or video feed, but the overall picture or sound could still be intelligible.

## The UDP Classes

*Java's support for UDP is contained in two classes, `java.net.DatagramSocket` and `java.net.DatagramPacket`. A `DatagramSocket` is used to send and receive `DatagramPackets`. Since UDP is connectionless, streams are not used. You must fit your data into packets of no more than about 60,000 bytes. You can manually split the data across multiple packets if necessary.*

### DatagramSocket

This class represents a socket for sending and receiving datagram packets.

*A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.*

```
DatagramSocket ds = new DatagramSocket(3000);
```

The Above Constructor is use to send Data at Specified port.

### DatagramPacket

*The `DatagramPacket` class is a wrapper for an array of bytes from which data will be sent or into which data will be received. It also contains the address and port to which the packet will be sent.*

#### Constructors

```
public DatagramPacket(byte[] data, int length)
public DatagramPacket(byte[] data, int length, InetAddress host, int port)
```

You construct a `DatagramPacket` object by passing an array of bytes and the number of those bytes to send to the `DatagramPacket()` constructor like this:

#### Methods

You can retrieve these items with these four get methods:

```
public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()
```

#### Here This Example Receive Packets Send By send.java program

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class receive {
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket(3000);
        byte[] buf = new byte[1024];
        DatagramPacket dp = new DatagramPacket(buf, 1024);
        ds.receive(dp);
        String strRecv = new String(dp.getData(), 0, dp.getLength()) + " from "
            + dp.getAddress().getHostAddress() + ":" + dp.getPort();
        System.out.println(strRecv);
    }
}
```

```
        ds.close();
    }
}
```

#### Here this program send data to Host 127.0.0.1 to portno 3000

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class send {
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket();
        String str = "hello world";
        InetAddress ia = InetAddress.getByName("127.0.0.1");
        DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ia, 3000);
        ds.send(dp);
        ds.close();
    }
}
```

#### Difference between Tcp Socket and UDP Socket

**1.TCP(Transmission Control Protocol).** TCP is a connection-oriented protocol, a connection can be made from client to server, and from then on any data can be sent along that connection.

- o **Reliable** - when you send a message along a TCP socket, you know it will get there unless the connection fails completely. If it gets lost along the way, the server will re-request the lost part. This means complete integrity, things don't get corrupted.
- o **Ordered** - if you send two messages along a connection, one after the other, you know the first message will get there first. You don't have to worry about data arriving in the wrong order.
- o **Heavyweight** - when the low level parts of the TCP "stream" arrive in the wrong order, resend requests have to be sent, and all the out of sequence parts have to be put back together, so requires a bit of work to piece together.

2. **UDP(User Datagram Protocol).** A simpler message-based connectionless protocol. With UDP you send messages (packets) across the network in chunks.

- o **Unreliable** - When you send a message, you don't know if it'll get there, it could get lost on the way.
- o **Not ordered** - If you send two messages out, you don't know what order they'll arrive in.
- o **Lightweight** - No ordering of messages, no tracking connections, etc. It's just fire and forget! This means it's a lot quicker, and the network card / OS have to do very little work to translate the data back from the packets.



# RMI



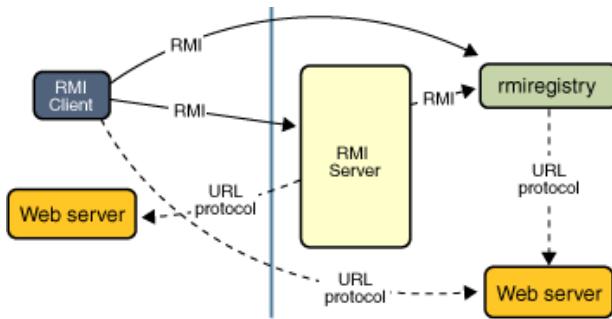
## *An Overview of RMI Applications*

Java RMI is a mechanism to allow the invocation of methods that reside on different Java Virtual Machines (JVMs). The JVMs may be on different machines or they could be on the same machine. In either case, the method runs in a different address space than the calling process.

**Java RMI is an object-oriented remote procedure call mechanism.**

- RMI applications often comprise two separate programs, a server and a client.
- A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
- A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

- RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.



**There are three entities involved in running a program that uses RMI:**

**Client:** this is the program that you write to access remote methods

**Server:** this is the program that you write to implement the remote methods - clients connect to the server and request that a method be executed. The remote methods to the client are local methods to the server.

**Object registry:** this is a program that you use.

The object registry runs on a known port (1099 by default)

A server, upon starting, registers its objects with a textual name with the object registry.

A client, before performing invoking a remote method, must first contact the object registry to obtain access to the remote object.

***To Implement RMI In Java One can Create Following Programs:***

- 1) **Remote Interface:** This program contains all the abstract methods which called by client remotely. This program use interface instead of class which extends the `java.rmi.Remote` interface which contains all remote features. Any method that can be remotely invoked in Java/RMI may throw a `java.rmi.RemoteException`.  
`java.rmi.RemoteException` is the superclass of many more RMI specific exception classes.

**Program: serverinterface.java**

```

import java.rmi.*;
public interface serverinterface extends Remote
{
    int getAmount(int qty) throws RemoteException;
}
  
```

- 2) **Implementing Remote interface:** in second program all abstract method which declared in interfaces are implemented in class. All methods which implement in class must throws `java.rmi.RemoteException`.  
This class must extends `UnicastRemoteObject` class to acquire the properties and behavior of Remote Object. The `UnicastRemoteObject` subclass exports the remote object to make it available for servicing incoming RMI calls.

**Program: serverclass.java**

```

import java.rmi.*;
import java.rmi.server.*;
public class serverclass extends UnicastRemoteObject implements serverinterface
{ private int rate;
    public serverclass(int r) throws RemoteException
    { rate=r;}
    public int getAmount(int qty)throws RemoteException
    { return(rate*qty);
    }
}

```

- 3) Client Registry: This program is used to register client by binding their names. To bind name one can use Naming.rebind() method, each client name bind with RemoteObjects which is further searched by client machine using method Naming.lookup() which returns reference of Remote object to client machine to call remote methods.

#### **Program:clientregistry.java**

```

import java.rmi.*;
public class clientregistry
{ public static void main(String arg[])
{ try
{ serverclass objsony=new serverclass(9000);
    serverclass objlg=new serverclass(8999);
    Naming.rebind("SONY",objsony);
    Naming.rebind("LG",objlg);
}
catch(Exception e)
{System.out.println(e); } }
}

```

- 4) Client Program: This program Calls the Remote Method.

```

Program:client.java
import java.rmi.*;
public class client
{ public static void main(String arg[])
{ try
{ String serurl="rmi://192.168.1.2/SONY";
    serverinterface i=(serverinterface)Naming.lookup(serurl);
    int amt=i.getAmount(4);
    System.out.println("Amount is:"+amt); }
catch(Exception e)
{System.out.println(e); } }
}

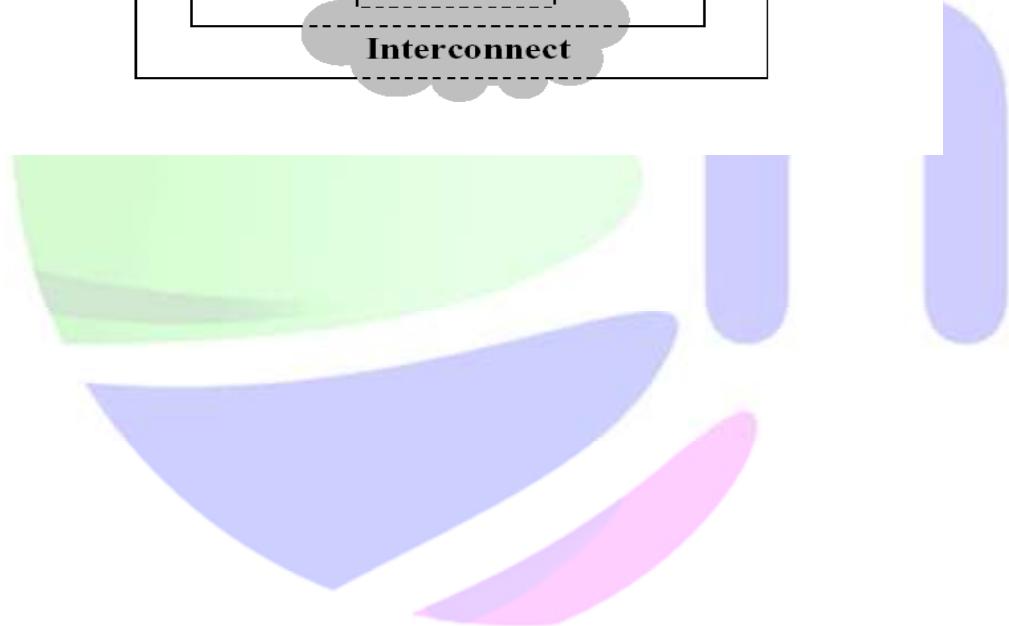
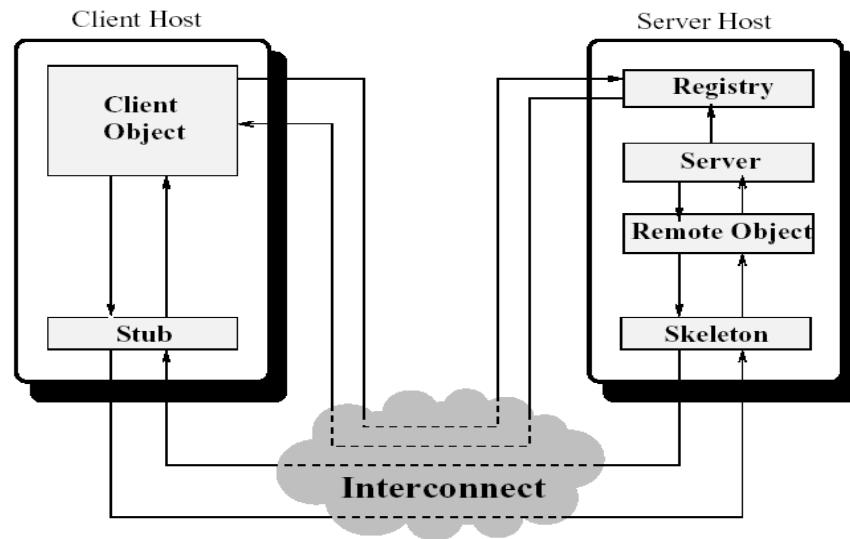
```

#### **To run the above program follow the following instruction.**

- **Compile All Programs**
- **Create a Stub using rmi compiler**  
rmic serverclass  
this will create stub class name serverclass\_stub.class which is further used to establish communication between client and server.
- **Start RMI server**  
rmiregistry
- **Register the clients on server**  
Java clientregistry
- **Run client program to invoke server methods**  
Java client

## How remoting works

To invoke a remote method, the client makes a call to the client proxy. The client side proxy packs the call parameters into a request message and invokes a wire protocol JRMP(in Java/RMI) to ship the message to the server. At the server side, the wire protocol delivers the message to the server side stub. The server side stub then unpacks the message and calls the actual method on the object Java/RMI, the client stub is called the **stub or proxy** and the server stub is called **skeleton**.



# Events



## AWT - Events

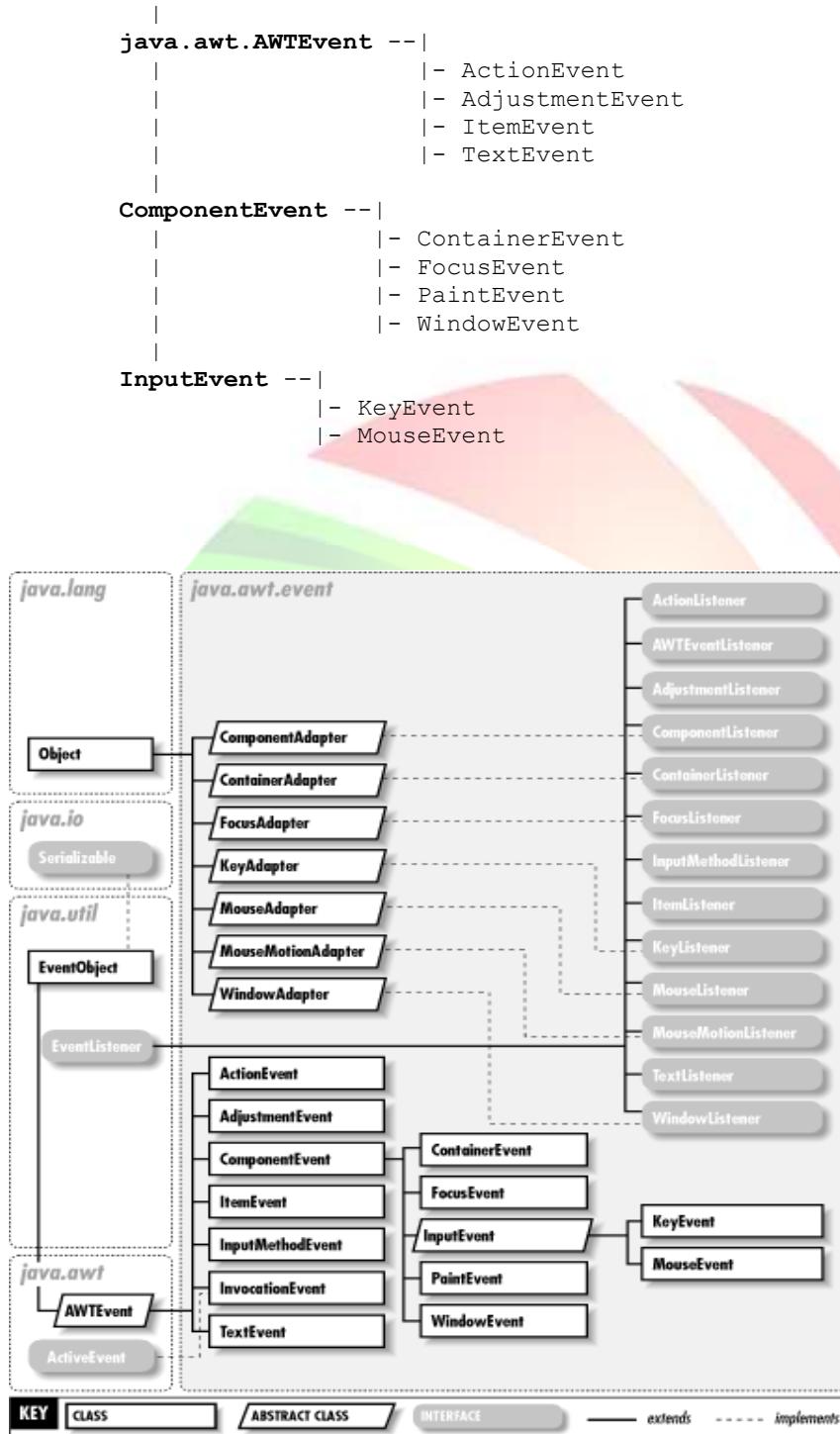
Java uses since the 1.1 version an event delegation models that allows you to designate any object as a listener for component's event. A component may have multiple listeners for any event type. All listener must implement the appropriate interface. If the interface defines more than one method, the listener may extend the appropriate adapter class.

There are two types of events; low-level events and semantic events. The AWT's semantic events are:

- ActionEvent
- AdjustmentEvent
- ItemEvent
- TextEvent

## Event Class Hierarchy

```
java.util.EventObject  
|
```



## Delegation of Event Handling to Listener Objects

An event listener is an object to which a component has delegated the task of handling a particular kind of event. When the component experiences input, an event of the appropriate type is constructed; the event is then passed as the parameter to a method call on the listener. A listener must implement the interface that contains the event-handling method.

The steps to support event handling using this model goes as follows:

1. Create a listener class that implements the desired listener interface.
2. Construct the component.
3. Construct an instance of the listener class.
4. Call addXXXListener() on the component, passing in the listener object.

## Introduction

There are many types of events that are generated by your AWT Application. These events are used to make the application more effective and efficient. Generally, there are twelve types of event are used in Java AWT. These are as follows :

1. ActionEvent
2. AdjustmentEvent
3. ComponentEvent
4. ContainerEvent
5. FocusEvent
6. InputEvent
7. ItemEvent
8. KeyEvent
9. MouseEvent
10. PaintEvent
11. TextEvent
12. WindowEvent

These are twelve mentioned events are explained as follows :

1. **ActionEvent:** This is the **ActionEvent** class extends from the **AWTEvent** class. It indicates the component-defined events occurred i.e. the event generated by the component like Button, Checkboxes etc. The generated event is passed to everyEventListener objects that receives such types of events using the `addActionListener()` method of the object.
2. **AdjustmentEvent:** This is the **AdjustmentEvent** class extends from the **AWTEvent** class. When the Adjustable Value is changed then the event is generated.
3. **ComponentEvent:** **ComponentEvent** class also extends from the **AWTEvent** class. This class creates the low-level event which indicates if the object moved, changed and its states (visibility of the object). This class only performs the notification about the state of the object. The **ComponentEvent** class performs like root class for other component-level events.
4. **ContainerEvent:** The **ContainerEvent** class extends from the **ComponentEvent** class. This is a low-level event which is generated when container's contents changes because of addition or removal of a components.
5. **FocusEvent:** The **FocusEvent** class also extends from the **ComponentEvent** class. This class indicates about the focus where the focus has gained or lost by the object. The generated event is passed to every objects that is registered to receive such type

of events using the `addFocusListener()` method of the object.

6. **InputEvent:** The **InputEvent** class also extends from the **ComponentEvent** class. This event class handles all the component-level input events. This class acts as a root class for all component-level input events.
7. **ItemEvent:** The **ItemEvent** class extends from the **AWTEvent** class. The **ItemEvent** class handles all the indication about the selection of the object i.e. whether selected or not. The generated event is passed to every **ItemListener** objects that is registered to receive such types of event using the `addItemListener()` method of the object.
8. **KeyEvent:** **KeyEvent** class extends from the **InputEvent** class. The **KeyEvent** class handles all the indication related to the key operation in the application if you press any key for any purposes of the object then the generated event gives the information about the pressed key. This type of events check whether the pressed key left key or right key, 'A' or 'a' etc.
9. **MouseEvent:** **MouseEvent** class also extends from the **InputEvent** class. The **MouseEvent** class handle all events generated during the mouse operation for the object. That contains the information whether mouse is clicked or not if clicked then checks the pressed key is left or right.
10. **PaintEvent:** **PaintEvent** class also extends from the **ComponentEvent** class. The **PaintEvent** class only ensures that the `paint()` or `update()` are serialized along with the other events delivered from the event queue.
11. **TextEvent:** **TextEvent** class extends from the **AWTEvent** class. **TextEvent** is generated when the text of the object is changed. The generated events are passed to every **TextListener** object which is registered to receive such type of events using the `addTextListener()` method of the object.
12. **WindowEvent :** **WindowEvent** class extends from the **ComponentEvent** class. If the window or the frame of your application is changed (Opened, closed, activated, deactivated or any other events are generated), **WindowEvent** is generated.

### AWTEvent

Most of the times every event-type has **Listener interface as Events** subclass the **AWTEvent class**. However, **PaintEvent** and **InputEvent** don't have the Listener interface because only the **paint() method** can be overridden with **PaintEvent** etc.

### Low-level Events

A **low-level input or window operation** is represented by the **Low-level events**. Types of Low-level events are **mouse movement, window opening, a key press** etc. For example, three events are generated by typing the letter 'A' on the Keyboard one for releasing, one for pressing, and one for typing. The different type of **low-level events and operations that generate each event** are show below in the form of a table.

|                       |                                                                                   |
|-----------------------|-----------------------------------------------------------------------------------|
| <b>FocusEvent</b>     | Used for Getting/losing focus.                                                    |
| <b>MouseEvent</b>     | Used for entering, exiting, clicking, dragging, moving, pressing, or releasing.   |
| <b>ContainerEvent</b> | Used for Adding/removing component.                                               |
| <b>KeyEvent</b>       | Used for releasing, pressing, or typing (both) a key.                             |
| <b>WindowEvent</b>    | Used for opening, deactivating, closing, Iconifying, deiconifying, really closed. |
| <b>ComponentEvent</b> | Used for moving, resizing, hiding, showing.                                       |

## Semantic Events

The interaction with GUI component is represented by the **Semantic events** like changing the text of a text field, selecting a button etc. The different events generated by different components is shown below.

|                        |                          |
|------------------------|--------------------------|
| <b>ItemEvent</b>       | Used for state changed.  |
| <b>ActionEvent</b>     | Used for do the command. |
| <b>TextEvent</b>       | Used for text changed.   |
| <b>AdjustmentEvent</b> | Used for value adjusted. |

## Event Sources

If a component is an event source for something then the same happens with its subclasses. The different event sources are represented by the following table.

| <b>Low-Level Events</b> |                                                                                           |
|-------------------------|-------------------------------------------------------------------------------------------|
| Window                  | WindowListener                                                                            |
| Container               | ContainerListener                                                                         |
| Component               | ComponentListener<br>FocusListener<br>KeyListener<br>MouseListener<br>MouseMotionListener |

| <b>Semantic Events</b>     |                    |
|----------------------------|--------------------|
| Scrollbar                  | AdjustmentListener |
| TextArea<br>TextField      | TextListener       |
| Button<br>List<br>MenuItem | ActionListener     |

|                  |              |
|------------------|--------------|
| TextField        |              |
| Choice           |              |
| Checkbox         |              |
| Checkbox         | ItemListener |
| CheckboxMenuItem |              |
| List             |              |

## Event Listeners

Every **listener interface** has at least one event type. Moreover, it also contains a method for each type of event the event class incorporates. For example as discussed earlier, the **KeyListener** has three methods, one for each type of event that the **KeyEvent** has: **keyTyped()**, **keyPressed()**, and **keyReleased()**.

The Listener interfaces and their methods are as follow:

| Interface          | Methods                                   |
|--------------------|-------------------------------------------|
| WindowListener     | windowActivated(WindowEvent e)            |
|                    | windowDeiconified(WindowEvent e)          |
|                    | windowOpened(WindowEvent e)               |
|                    | windowClosed(WindowEvent e)               |
|                    | windowClosing(WindowEvent e)              |
|                    | windowIconified(WindowEvent e)            |
|                    | windowDeactivated(WindowEvent e)          |
| ActionListener     | actionPerformed(ActionEvent e)            |
| AdjustmentListener | adjustmentValueChanged(AdjustmentEvent e) |
| MouseListener      | mouseClicked(MouseEvent e)                |
|                    | mouseEntered(MouseEvent e)                |
|                    | mouseExited(MouseEvent e)                 |
|                    | mousePressed(MouseEvent e)                |
|                    | mouseReleased(MouseEvent e)               |
| FocusListener      | focusGained(FocusEvent e)                 |
| FocusListener      | focusLost(FocusEvent e)                   |
| ItemListener       | itemStateChanged(ItemEvent e)             |
| KeyListener        | keyReleased(KeyEvent e)                   |
| KeyListener        | keyTyped(KeyEvent e)                      |
| ComponentListener  | keyPressed(KeyEvent e)                    |
|                    | componentHidden(ComponentEvent e)         |
|                    | componentMoved(ComponentEvent e)          |
|                    | componentShown(ComponentEvent e)          |
| ComponentListener  | componentResized(ComponentEvent e)        |
|                    | mouseMoved(MouseEvent e)                  |
|                    | mouseDragged(MouseEvent e)                |
| TextListener       | textValueChanged(TextEvent e)             |
| ContainerListener  | componentAdded(ContainerEvent e)          |
|                    | componentRemoved(ContainerEvent e)        |

## Adapter Classes

There are some event listeners that have multiple methods to implement. That is some of the listener interfaces contain more than one method. For instance, the **MouseListener** interface contains five methods such as **mouseClicked**, **mousePressed**, **mouseReleased** etc. If you want to use only one method out of these then also you will have to implement all of them. Thus, the methods which you do not want to care about can have empty bodies. To avoid such thing, we have **adapter class**.

**Adapter classes** help us in avoiding the implementation of the empty method bodies. Generally an adapter class is there for each listener interface having more than one method. For instance, the **MouseAdapter** class implements the **MouseListener** interface. An **adapter class** can be used by **creating a subclass of it** and then **overriding the methods** which are of use only. Hence avoiding the implementation of all the methods of the listener interface.

**The Following is a list of Adapter classes and Listener Interfaces In Java:**

| Event Category | Interface Name      | Adapter Name       | Method                                  |
|----------------|---------------------|--------------------|-----------------------------------------|
| Window         | Window Listener     | Window Adapter     | Void windowClosing (WindowEvent e)      |
|                |                     |                    | Void windowOpened (WindowEvent e)       |
|                |                     |                    | Void windowActivated (WindowEvent e)    |
|                |                     |                    | Void windowDeactivated (WindowEvent e)  |
|                |                     |                    | Void windowClosed (WindowEvent e)       |
|                |                     |                    | Void windowIconified (WindowEvent e)    |
|                |                     |                    | Void windowDeiconified (WindowEvent e)  |
| Action         | ActionListener      |                    | Void actionPerformed(ActionEvent e)     |
| Item           | ItemListener        |                    | Void itemStateChanged(ItemEvent e)      |
| Mouse Motion   | MouseMotionListener | MouseMotionAdapter | Void mouseDragged(MouseEvent e)         |
|                |                     |                    | Void mouseMoved(MouseEvent e)           |
| Mouse Button   | MouseListener       | MouseAdapter       | Void mousePressed(MouseEvent e)         |
|                |                     |                    | Void mouseReleased(MouseEvent e)        |
|                |                     |                    | Void mouseEntered(MouseEvent e)         |
|                |                     |                    | Void mouseClicked(MouseEvent e)         |
|                |                     |                    | Void mouseExited(MouseEvent e)          |
| Key            | KeyListener         | KeyAdapter         | Void keyPressed(KeyEvent e)             |
|                |                     |                    | Void keyReleased(KeyEvent e)            |
|                |                     |                    | Void keyTyped(KeyEvent e)               |
| Focus          | FocusListener       |                    | Void focusGained(FocusEvent e)          |
|                |                     |                    | Void focusLost(FocusEvent e)            |
| Component      | ComponentListener   | ComponentAdapter   | Void componentMoved(ComponentEvent e)   |
|                |                     |                    | Void componentResized(ComponentEvent e) |
|                |                     |                    | Void componentHidden(ComponentEvent e)  |
|                |                     |                    | Void componentShown(ComponentEvent e)   |

## Introduction to servlets

### What does a servlet do?

A servlet is a server-side software program, written in Java code, that handles messaging between a client and server. The Java Servlet API defines a standard interface for the request and response messages so your servlets can be portable across platforms and across different Web application servers.

Servlets can respond to client requests by dynamically constructing a response that is sent back to the client. For example, in this tutorial you'll write a servlet to respond to an HTTP request.

Because servlets are written in the Java programming language, they have access to the full set of Java APIs. This makes them ideal for implementing complex business application logic and especially for accessing data elsewhere in the enterprise. A single servlet can be invoked multiple times to serve requests from multiple clients. A servlet can handle multiple requests concurrently and can synchronize requests.

Servlets can forward requests to other servers and servlets.

### How does a servlet work?

A servlet runs inside a Java-enabled application server. Application servers are a special kind of Web server; they extend the capabilities of a Web server to handle requests for servlets, enterprise beans, and Web applications. There is a distinct difference between a Web server and an application server. While both can run in the same machine, the Web server runs client code such as applets and the application server runs the servlet code.

The server itself loads, executes, and manages servlets. The server uses a Java bytecode interpreter to run Java programs; this is called the Java Virtual Machine (JVM).

The basic flow is this:

1. The client sends a request to the server.
2. The server instantiates (loads) the servlet and creates a thread for the servlet process. Note the servlet is loaded upon the first request; it stays loaded until the server shuts down.
3. The server sends the request information to the servlet.
4. The servlet builds a response and passes it to the server.
5. The server sends the response back to the client.

The servlet dynamically constructs the response using information from the client request, plus data gathered from other sources if needed. Such sources could be other servlets, shared objects, resource files, and databases. Shared resources, for example, could include in-memory data such as instance or class variables and external objects such as files, database connections, and network connections.

### How are servlets different from CGI programs?

Common gateway interface (CGI) programs are also used to construct dynamic Web content in response to a request. But servlets have several advantages over CGI. Servlets provide a component-based, platform-independent method for building Web applications, without the performance limitations of CGI programs. Servlets are:

\* **Portable across platforms and across different Web servers.** Servlets enable you to do server-side programming without writing to platform-specific APIs; the Java Servlet API is a standard Java extension.

\* **Persistent.** A servlet remains in memory once loaded, which means it can maintain system resources -- such as a database connection -- between requests.

- \* **Efficient.** When a client makes multiple calls to a servlet, the server creates and loads the servlet only once. Each additional call performs only the business logic processing. CGI processes are loaded with each request, which slows performance. In addition, the JVM uses lightweight Java threads to handle servlet requests, rather than a weighty operating system process used by CGI.
- \* **Able to separate presentation from business logic.** This makes it easier to split a project into distinct parts for easier development and maintenance.
- \* **Able to access a library of HTTP-specific calls** and to benefit from the continuing development of the Java language itself.

## What is the Java Servlet API?

The Java Servlet API is a set of classes that define a standard interface between a Web client and a Web servlet. In essence, the API encases requests as objects so the server can pass them to the servlet; the responses are similarly encapsulated so the server can pass them back to a client.

The Java Servlet API has two packages. `javax.servlet` contains classes to support generic protocol-independent servlets, and `javax.servlet.http` includes specific support for the HTTP protocol.

## On the Web: HTTP servlets

The `Servlet` interface class is the central abstraction of the Java Servlet API. This class defines the methods that manage the servlet and its communications with clients.

To write an HTTP servlet for use on the Web, use the `HttpServlet` class.

- \* A client request to a servlet is represented by an `HttpServletRequest` object. This object encapsulates the communication from the client to the server. It can contain information about the client environment and any data to be sent to the servlet from the client.
- \* The response from the servlet back to the client is represented by an `HttpServletResponse` object. This is often the dynamically generated response, such as an HTML page, and is built with data from the request and from other sources accessed by the servlet.

One example of a simple request-response scenario is to pass a parameter by appending it to a URL. This is a simple way for Web administrators to track a session, especially in cases where a client does not accept cookies. This tutorial demonstrates this scenario with our `Redirect` servlet, described in Section 7. The `Redirect` servlet takes a target URL and a referring origin as input and redirects the Web browser to the target URL.

## Key methods for processing HTTP servlets

Your subclass of `HttpServlet` must override at least one method. Typically, servlets override the `doGet` or `doPost` method. GET requests are typical browser requests for Web pages, issued when a user types a URL or follows a link. POST requests are generated when a user submits an HTML form that specifies post. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers. The same servlet can handle both GET and POST by having `doGet` call `doPost`, or vice versa.

Other commonly used methods include:

- \* `service`, the lowest common denominator method for implementing a servlet; most people use `doGet` or `doPost`
- \* `doPut`, for HTTP PUT requests
- \* `doDelete`, for HTTP DELETE requests
- \* `init` and `destroy`, to manage resources that are held for the life of the servlet
- \* `getServletInfo`, which the servlet uses to provide information about itself

## The role of the server

### A home for servlets

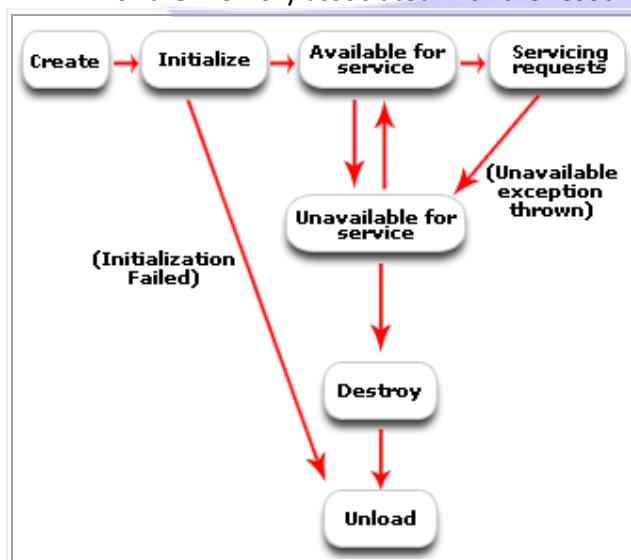
Servlet code follows the standard API, but the servers in which the code runs vary from free servlet engines to commercial servers. Typically what differentiates the free from the fee servers is the complexity of setup and configuration, compatibility with corollary products, and technical support. If you're just learning servlets, it may be useful to download a free server or servlet engine for your development and testing purposes.

A servlet engine runs within the application server and manages the servlet. It loads and initializes the servlet, passes requests and responses to the servlet and client, manages multiple instances of a servlet, and acts as a request dispatcher. It destroys the servlet at the end of its useful life. Servlets are unloaded and removed from memory when the server shuts down.

1. **Loading and Instantiation:** The servlet container loads the servlet during startup or when the first request is made.
2. **Initialization:** After creating the instances, the servlet container calls the init() method and passes the servlet initialization parameters to the init() method. The init() must be called by the servlet container before the servlet can service any request. The initialization parameters persist until the servlet is destroyed. The init() method is called only once throughout the life cycle of the servlet.

The servlet will be available for service if it is loaded successfully otherwise the servlet container unloads the servlet.

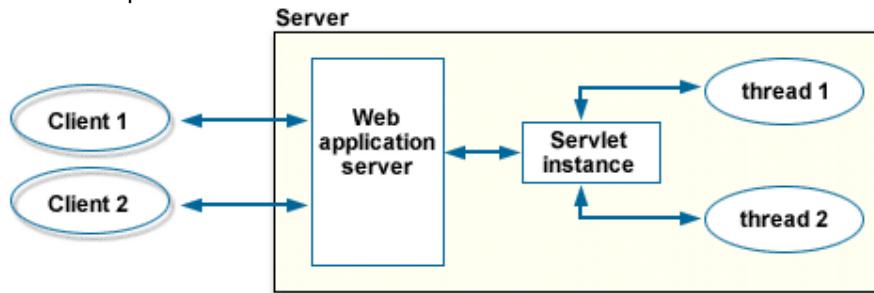
3. **Servicing the Request:** After successfully completing the initialization process, the servlet will be available for service. Servlet creates separate threads for each request. The servlet container calls the service() method for servicing any request. The service() method determines the kind of request and calls the appropriate method (doGet() or doPost()) for handling the request and sends response to the client using the methods of the response object.
4. **Destroying the Servlet:** If the servlet is no longer needed for servicing any request, the servlet container calls the destroy() method. Like the init() method this method is also called only once throughout the life cycle of the servlet. Calling the destroy() method indicates to the servlet container not to send any request for service and the servlet releases all the resources associated with it. Java Virtual Machine claims for the memory associated with the resources for garbage collection.



## Life Cycle of a Servlet

### Handling multithreading

When there are simultaneous requests to a servlet, the server handles each client request by creating a new thread for each request. There is a single thread per client/server request; this thread is used again and again each time the same client makes a request to the servlet.



For example:

- \* A servlet is loaded by the Web application server. There is one instance of a servlet object at a time, and it remains persistent for the life of the servlet.
- \* Two browser clients request the services of the servlet. The server creates a handler thread, for each request, against the instance object. Each thread has access to variables that were initialized when the servlet was loaded.
- \* Each thread handles its own requests. The server sends responses back to the appropriate client.

### Request/response handling

Note that the request/response handling done inside the servlet (with the `HttpServletRequest` and `HttpServletResponse` objects) is different from the request handling done by the Web server. The server handles communication with the external entities such as the client by passing the input, then the output, to the servlet.

## HttpServlet class and its Method

`HttpServlet` class is an abstract class defined in "javax.servlet.http" package and is sub class of `GenericServlet` class defined in "javax.servlet" package. `GenericServlet` class is used to define a Generic, protocol-independent servlet but `HttpServlet` class provides an HTTP protocol specific implementation. If your servlet has to work with HTTP protocol then your servlet needs to extend `HttpServlet` class. This class provides various `doXXX()` methods which can be overridden according to the type of request. This class also provides `service()` methods that can also be overridden but we don't need to do so because it calls the appropriate `doXXX()` method according to the method of request. Generally, `doGet()` or `doPost()` methods are needed to be overridden to handle the GET or POST requests.

A servlet is managed through a well defined life cycle that defines how it is loaded and instantiated, is initialized, handles requests from clients, and is taken out of service. This life cycle is expressed in the API by the `init`, `service`, and `destroy` methods of the `javax.servlet.Servlet`

interface that all servlets must implement directly or indirectly through the GenericServlet or HttpServlet abstract classes.

### **Servlet class loading and instantiation.**

The servlet container is responsible for loading and instantiating servlets. The loading and instantiation can occur when the container is started, or delayed until the container determines the servlet is needed to service a request.

When the servlet engine is started, needed servlet classes must be located by the servlet container. The servlet container loads the servlet class using normal Java class loading facilities. The loading may be from a local file system, a remote file system, or other network services.

After loading the Servlet class, the container instantiates it for use.

### **Servlet class initialization.**

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data, initialize costly resources (such as JDBC connections), and perform other one-time activities. The container initializes the servlet instance by calling the init method of the Servlet interface with a unique (per servlet declaration) object implementing the ServletConfig interface.

```
public void init(ServletConfig config) throws ServletException;
```

This configuration object allows the servlet to access name-value initialization parameters from the Web application's configuration information. The configuration object also gives the servlet access to an object (implementing the ServletContext interface) that describes the servlet's runtime environment.

During initialization, the servlet instance can throw an UnavailableException or a ServletException. In this case, the servlet must not be placed into active service and must be released by the servlet container. The destroy method is not called as it is considered unsuccessful initialization.

A new instance may be instantiated and initialized by the container after a failed initialization. The exception to this rule is when an UnavailableException indicates a minimum time of unavailability, and the container must wait for the period to pass before creating and initializing a new servlet instance.

### **Request handling.**

After a servlet is properly initialized, the servlet container may use it to handle client requests. Requests are represented by request objects of type HttpServletRequest. The servlet fills

out response to requests by calling methods of a provided object of type `ServletResponse`.  
These

In the case of an HTTP request, the objects provided by the container are of types

## **HttpServletRequest and HttpServletResponse.**

```
protected void service(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException;
```

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
```

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
```

Note that a servlet instance placed into service by a servlet container may handle NO requests during its lifetime.

### **End of service.**

The servlet container is not required to keep a servlet loaded for any particular period of time. A servlet instance may be kept active in a servlet container for a period of milliseconds, for the lifetime of the servlet container (which could be a number of days, months, or years), or any amount of time in between.

When the servlet container determines that a servlet should be removed from service, it calls the `destroy` method of the `Servlet` interface to allow the servlet to release any resources it is using and save any persistent state. For example, the container may do this when it wants to conserve memory resources, or when it is being shut down.

```
public void destroy();
```

Before the servlet container calls the `destroy` method, it must allow any threads that are currently running in the `service` method of the servlet to complete execution, or exceed a server-defined time limit.

Once the `destroy` method is called on a servlet instance, the container may not route other requests to that instance of the servlet. If the container needs to enable the servlet again, it must do so with a new instance of the servlet's class.

After the `destroy` method completes, the servlet container must release the servlet instance so that it is eligible for garbage collection.

## HttpServletRequest interface

HttpServletRequest interface is defined in "javax.servlet.http" and is used to retrieve parameters from the request, HTTP request header information, cookies from the request, getting session object to maintain session for the request etc. This interface extends the ServletRequest interface to provide request information for HTTP servlets. The HttpServletRequest object is created by the servlet container and is passed as an argument to the servlet's service, doGet, doPost methods etc.

### **HTTP Protocol Parameters.**

Request parameters for the servlet are the strings sent by the client to a servlet container as part of its request. When the request is an HttpServletRequest object, the container populates the parameters from the URI query string and POST-ed data.

The parameters are stored as a set of name-value pairs. Multiple parameter values CAN exist for any given parameter name. The following methods of the ServletRequest interface are available to access parameters:

- `getParameter`

Returns the value of a request parameter as a String, or null if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

You should only use this method when you are sure the parameter has only ONE value. If the parameter might have MORE than one value, use

`getParameterValues(String).`

If you use this method with a multivalued parameter, the value returned is equal to the FIRST value in the array returned by `getParameterValues`.

If the parameter data was sent in the request body, such as occurs with an HTTP POST request, then reading the body directly via `getInputStream()` or `getReader()` can interfere with the execution of this method.

```
public java.lang.String getParameter(java.lang.String name);
```

- `getParameterNames`

Returns an Enumeration of String objects containing the names of the parameters contained in this request. If the request has no parameters, the method returns an EMPTY Enumeration.

```
public java.util.Enumeration getParameterNames();
```

```
public String[] getParameterNames();
```

- `getParameterValues`

Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist. If the parameter has a single value, the array has a length of 1.

```
public java.lang.String[] getParameterValues(java.lang.String name);
```

- `getParameterMap`

Returns an immutable java.util.Map containing parameter names as keys and parameter values as map values. The keys in the parameter map are of type String. The values in the parameter map are of type String array.

```
public java.util.Map<String, String> getParameterMap();
```

## Cookies

The HttpServletRequest interface provides the getCookies method to obtain an array of cookies that are present in the request. This method returns null if no cookies were sent.

The cookies are data sent from the client to the server on every request that the client makes. Typically, the only information that the client sends back as part of a cookie is the cookie name and the cookie value. Other cookie attributes that can be set when the cookie is sent to the browser, such as comments, are not typically returned. Several cookies might have the same name but different path attributes.

```
public Cookie[] getCookies();
```

## The HttpServletResponse interface

The HttpServletResponse interface extends the ServletResponse interface and provides the functionalities to the classes for accessing the HTTP response. The servlet container creates an HttpServletRequest object and an HttpServletResponse object, and passes them as arguments to the servlet's service methods such as doGet, doPost, etc.

### Content type.

The charset for the MIME body response can be specified explicitly using the setContentType(String) method. Explicit specifications take precedence over implicit specifications. The setContentType method MUST be called BEFORE getWriter and BEFORE committing the response for the character encoding to be used.

There are 2 ways to define content type:

- void setContentType(String);

### Acquire a text stream.

To send CHARACTER data, use the PrintWriter object returned by `ServletResponse.getWriter()`

```
public java.io.PrintWriter getWriter() throws IOException;
```

Returns a PrintWriter object that can send character text to the client. The PrintWriter uses the character encoding returned by `getCharacterEncoding()`. Calling `flush()` on the PrintWriter commits the response.

### Acquire a binary stream.

`ServletResponse.getOutputStream()` provides an output stream for sending BINARY data to the client. A `ServletOutputStream` object is normally retrieved via this method.

```
public ServletOutputStream getOutputStream() throws IOException;
```

The servlet container does NOT encode the binary data.

Calling `flush()` on the `ServletOutputStream` commits the response. Either this method or `getWriter()` may be called to write the body, NOT BOTH.

### Redirect an HTTP request to another URL.

The `HttpServletResponse.sendRedirect` method will set the appropriate headers and content body to redirect the client to a different URL. It is legal to call this method with a relative URL path, however the underlying container must translate the relative path to a fully qualified URL for transmission back to the client. If a partial URL is given and, for whatever reason, cannot be converted into a valid URL, then this method must throw an `IllegalArgumentException`.

```
public interface HttpServletResponse extends javax.servlet.ServletResponse {  
  
    public void sendRedirect(java.lang.String location) throws IOException;  
  
}
```

Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container must convert the relative URL to an absolute URL before sending the response to the client. If the location is relative without a leading '/' the container interprets it as relative to the current request URI. If the location is relative with a leading '/' the container interprets it as relative to the servlet container root. If the response has already been committed, this method throws an `IllegalStateException`. After using this method, the response should be considered to be committed and should not be written to.

This method will have the side effect of committing the response, if it has not already been committed, and terminating it. No further output to the client should be made by the servlet after these methods are called. If data is written to the response after this method are called, the data is ignored.

If data has been written to the response buffer, but not returned to the client (i.e. the response is not committed), the data in the response buffer must be cleared and replaced with the data set by these methods. If the response is committed, this method must throw an `IllegalStateException`.

#### Add cookies to the response.

The servlet sends cookies to the browser by using the `HttpServletResponse.addCookie(Cookie)` method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.

```
public void addCookie(Cookie cookie);
```

Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.

## Cookies

### **Cookie Class**

In JSP cookie are the object of the class **`javax.servlet.http.Cookie`**. This class is used to creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a

client, so cookies are commonly used for session management. A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

The `getCookies()` method of the request object returns an array of `Cookie` objects. Cookies can be constructed using the following code:

```
Cookie(java.lang.String name, java.lang.String value)
```

`Cookie` objects have the following methods.

| Method                          | Description                                                                                                                                                                                                                                                                                   |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getComment()</code>       | Returns the comment describing the purpose of this cookie, or null if no such comment has been defined.                                                                                                                                                                                       |
| <code>getMaxAge()</code>        | Returns the maximum specified age of the cookie.                                                                                                                                                                                                                                              |
| <code>getName()</code>          | Returns the name of the cookie.                                                                                                                                                                                                                                                               |
| <code>getPath()</code>          | Returns the prefix of all URLs for which this cookie is targeted.                                                                                                                                                                                                                             |
| <code>getValue()</code>         | Returns the value of the cookie.                                                                                                                                                                                                                                                              |
| <code>setComment(String)</code> | If a web browser presents this cookie to a user, the cookie's purpose will be described using this comment.                                                                                                                                                                                   |
| <code>setMaxAge(int)</code>     | Sets the maximum age of the cookie. The cookie will expire after that many seconds have passed. Negative values indicate the default behavior: the cookie is not stored persistently, and will be deleted when the user web browser exits. A zero value causes the cookie to be deleted       |
| <code>setPath(String)</code>    | This cookie should be presented only with requests beginning with this URL.                                                                                                                                                                                                                   |
| <code>setValue(String)</code>   | Sets the value of the cookie. Values with various special characters (white space, brackets and parentheses, the equals sign, comma, double quote, slashes, question marks, the "at" sign, colon, and semicolon) should be avoided. Empty values may not behave the same way on all browsers. |

### Example Using Cookies

Cookies are small bits of information that a Web server sends to a browser and that the browser returns unchanged after visiting the same page. A Class **javax.servlet.http.Cookie**, represents the cookie. **HttpServletRequest** and **HttpServletResponse** interfaces have methods for getting and setting the cookies. You can create cookie, read cookie and send cookie to the client browser. You can create cookie by calling the cookie Constructor.

To read cookies, call `request.getCookies()` method which returns an array of cookie objects. To send cookie, a servlet create cookie and add the cookie to the response header with method `response.addCookie(cookie)`.

Here is the code of `SendCookie.java`

```
import java.io.*;
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
public class SendCookie extends HttpServlet
```

```

{
public void doGet( HttpServletRequest req, HttpServletResponse res )
throws IOException{
    int var, i ;
    String s ;
    Cookie [] cookie = req.getCookies() ;
    Cookie countCookie = null ;
    for( i = 0; i < cookie.length; i++ ){
if( cookie[i].getName().equals("Mycookie") ){
    countCookie = cookie[i] ;
}
}
if( countCookie != null ){
s = countCookie.getValue() ;
if( s != null ){
var = Integer.parseInt( s ) ;
}
else{
    var = 0 ;
}
}
else{
try{
    countCookie = new Cookie( "Mycookie", null ) ;
}catch( IllegalArgumentException exception ){}
var = 0 ;
}
var++ ;
Integer ol = new Integer( var ) ;
countCookie.setValue( ol.toString() ) ;
res.addCookie( countCookie ) ;
PrintWriter out = res.getWriter() ;
out.write( "You've visited the page" + var+ " times" ) ;
}
}
}

```

## Session Tracking

As we know that the Http is a *stateless* protocol, means that it can't persist the information. It always treats each request as a new request. In Http client makes a connection to the server, sends the request., gets the response, and closes the connection.

In session management client first make a request for any servlet or any page, the container receives the request and generate a unique session ID and gives it back to the client along with the response. This ID gets stores on the client machine. Thereafter when the client request again sends a request to the server then it also sends the session Id with the request. There the container sees the Id and sends back the request.

Session Tracking can be done in three ways:

1. **Hidden Form Fields:** This is one of the way to support the session tracking. As we know by the name, that in this fields are added to an HTML form which are not displayed in the client's request. The hidden form field are sent back to the server when the form is submitted. In hidden form fields the html entry will be like this :

- <input type = "hidden" name = "name" value="">. This means that when you submit the form, the specified name and value will be get included in get or post method. In this session ID information would be embedded within the form as a hidden field and submitted with the Http POST command.
2. **URL Rewriting:** This is another way to support the session tracking. **URLRewriting** can be used in place where we don't want to use cookies. It is used to maintain the session. Whenever the browser sends a request then it is always interpreted as a new request because http protocol is a stateless protocol as it is not persistent. Whenever we want that our request object to stay alive till we decide to end the request object then, there we use the concept of session tracking. In session tracking firstly a session object is created when the first request goes to the server. Then server creates a token which will be used to maintain the session. The token is transmitted to the client by the response object and gets stored on the client machine. By default the server creates a cookie and the cookie get stored on the client machine.
3. **Cookies:** When cookie based session management is used, a token is generated which contains user's information, is sent to the browser by the server. The cookie is sent back to the server when the user sends a new request. By this cookie, the server is able to identify the user. In this way the session is maintained. Cookie is nothing but a name-value pair, which is stored on the client machine. By default the cookie is implemented in most of the browsers. If we want then we can also disable the cookie. For security reasons, cookie based session management uses two types of cookies.

## HttpSession interface

HttpSession interface is defined in "javax.servlet.http" package and is used for the purpose of session tracking while working with servlets. Session tracking is a mechanism that is used to maintain state about a series of requests from the same user (that is, requests originating from the same browser) across some period of time. The session object can be found using `getSession()` method of the servlet request object. With the help of this interface the servlet can view and manipulate information about a session, for example, session id, creation time, last accessed time and can bind the objects containing user information to the session. Storing all these information to the session persists across series of requests from the same user.

### Method Index

| Method                                  | Description                                                                                                       |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>long getCreationTime()</code>     | Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.      |
| <code>String getId()</code>             | Returns a string containing the unique identifier assigned to this session.                                       |
| <code>long getLastAccessedTime()</code> | Returns the last time the client sent a request associated with this session, as the number of milliseconds since |

|                                               |                                                                                                                                |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
|                                               | midnight January 1, 1970 GMT, and marked by the time the container received the request.                                       |
| <code>int getMaxInactiveInterval()</code>     | Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. |
| <code>Object getValue(String)</code>          | <b>Deprecated.</b> As of Version 2.2, this method is replaced by <code>getAttribute</code> .                                   |
| <code>String[] getValueNames()</code>         | <b>Deprecated.</b> As of Version 2.2, this method is replaced by <code>getAttributeNames</code>                                |
| <code>void invalidate()</code>                | Invalidate this session then unbinds any objects bound to it.                                                                  |
| <code>boolean isNew()</code>                  | Returns <code>true</code> if the client does not yet know about the session or if the client chooses not to join the session.  |
| <code>void putValue(String, Object)</code>    | <b>Deprecated.</b> As of Version 2.2, this method is replaced by <code>setAttribute</code>                                     |
| <code>void removeValue(String)</code>         | <b>Deprecated.</b> As of Version 2.2, this method is replaced by <code>removeAttribute</code>                                  |
| <code>void setMaxInactiveInterval(int)</code> | Specifies the time, in seconds, between client requests before the servlet container will invalidate this session.             |

## RequestDispatcher mechanism

## RequestDispatcher description.

RequestDispatcher defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the RequestDispatcher object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

An object implementing the RequestDispatcher interface may be obtained via the following methods:

- `ServletContext.getRequestDispatcher(String path)`
- `ServletContext.getNamedDispatcher(String name)`
- `ServletRequest.getRequestDispatcher(String path)`

The `ServletContext.getRequestDispatcher` method takes a string argument describing a path within the scope of the `ServletContext`. This path must be relative to the root of the `ServletContext` and begin with a '/'. The method uses the path to look up a servlet, using the servlet path matching rules, wraps it with a `RequestDispatcher` object, and returns the resulting object. If no servlet can be resolved based on the given path, a `RequestDispatcher` is provided that returns the content for that path.

The `ServletContext.getNamedDispatcher` method takes a String argument indicating the NAME of a servlet known to the `ServletContext`. If a servlet is found, it is wrapped with a `RequestDispatcher` object and the object is returned. If no servlet is associated with the given name, the method must return null.

To allow `RequestDispatcher` objects to be obtained using relative paths that are relative to the path of the current request (not relative to the root of the `ServletContext`), the `ServletRequest.getRequestDispatcher` method is provided in the `ServletRequest` interface. The behavior of this method is similar to the method of the same name in the `ServletContext`. The servlet container uses information in the request object to transform the given relative path against the current servlet to a complete path. For example, in a context rooted at '/' and a request to /garden/tools.html, a request dispatcher obtained via `ServletRequest.getRequestDispatcher("header.html")` will behave exactly like a call to `ServletContext.getRequestDispatcher("/garden/header.html")`.

#### **RequestDispatcher creation and using.**

```
public class Dispatcher extends HttpServlet {  
    public void doGet(HttpServletRequest req, HttpServletResponse res) {  
        RequestDispatcher dispatcher =  
            request.getRequestDispatcher("/template.jsp");  
        if (dispatcher != null) dispatcher.forward(request, response);  
    }  
}
```

`forward` should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an `IllegalStateException`. Uncommitted output in the response buffer is automatically cleared before the forward.

```
public class Dispatcher extends HttpServlet {  
    public void doGet(HttpServletRequest req, HttpServletResponse res) {  
        RequestDispatcher dispatcher =  
            getServletContext().getRequestDispatcher("/banner");  
        if (dispatcher != null) dispatcher.include(request, response);  
    }  
}
```

Includes the content of a resource (servlet, JSP page, HTML file) in the response. In essence, this method enables programmatic server-side includes. The `ServletResponse` object has its path elements and parameters remain unchanged from the caller's. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.

```
package javax.servlet;
```

```
public void forward(ServletRequest request, ServletResponse response)  
throws ServletException, java.io.IOException;  
public void include(ServletRequest request, ServletResponse response)  
throws ServletException, java.io.IOException;
```

The `include` method of the `RequestDispatcher` interface may be called at ANY time. The target servlet of the `include` method has access to all aspects of the request object, but its use of

the response object is more limited. It can only write information to the ServletOutputStream or Writer of the response object and commit a response by writing content past the end of the response buffer, or by explicitly calling the flushBuffer method of the ServletResponse interface. It CANNOT set headers or call any method that affects the headers of the response. Any attempt to do so must be ignored.

The forward method of the RequestDispatcher interface may be called by the calling servlet ONLY when NO output has been committed to the client. If output data exists in the response buffer that has not been committed, the content must be cleared before the target servlet's service method is called. If the response has been committed, an IllegalStateException must be thrown.

The path elements of the request object exposed to the target servlet must reflect the path used to obtain the RequestDispatcher. The only exception to this is if the RequestDispatcher was obtained via the getNamedDispatcher method. In this case, the path elements of the request object must reflect those of the original request. Before the forward method of the RequestDispatcher interface returns, the response content MUST be sent and committed, and closed by the servlet container.

The ServletContext and ServletRequest methods that create RequestDispatcher objects using path information allow the optional attachment of query string information to the path. For example, a Developer may obtain a RequestDispatcher by using the following code:

```
String path = "/raisins.jsp?orderno=5";
RequestDispatcher rd = context.getRequestDispatcher(path);
rd.include(request, response);
```

Parameters specified in the query string used to create the RequestDispatcher take precedence over other parameters of the same name passed to the included servlet. The parameters associated with a RequestDispatcher are scoped to apply only for the duration of the include or forward call.

#### **Additional request-scoped attributes.**

Except for servlets obtained by using the getNamedDispatcher method, a servlet that has been invoked by another servlet using the include method of RequestDispatcher has access to the path by which it was invoked.

The following request attributes must be set:

- javax.servlet.include.request\_uri
- javax.servlet.include.context\_path
- javax.servlet.include.servlet\_path
- javax.servlet.include.path\_info
- javax.servlet.include.query\_string

These attributes are accessible from the included servlet via the `getAttribute` method on the `request` object and their values must be equal to the request URI, context path, servlet path, path info, and query string of the INCLUDED servlet, respectively. If the request is subsequently included, these attributes are replaced for that include.

If the included servlet was obtained by using the `getNamedDispatcher` method, these attributes MUST NOT be set.

Except for servlets obtained by using the `getNamedDispatcher` method, a servlet that has been invoked by another servlet using the `forward` method of `RequestDispatcher` has access to the path of the ORIGINAL request.

The following request attributes must be set:

- `javax.servlet.forward.request_uri`
- `javax.servlet.forward.context_path`
- `javax.servlet.forward.servlet_path`
- `javax.servlet.forward.path_info`
- `javax.servlet.forward.query_string`

The values of these attributes must be equal to the return values of the `HttpServletRequest` methods `getRequestURI`, `getContextPath`, `getServletPath`, `getPathInfo`, `getQueryString` respectively, invoked on the `request` object passed to the first servlet object in the call chain that received the request from the client.

These attributes are accessible from the forwarded servlet via the `getAttribute` method on the `request` object. Note that these attributes must always reflect the information in the original request even under the situation that multiple forwards and subsequent includes are called.

If the forwarded servlet was obtained by using the `getNamedDispatcher` method, these attributes must not be set.

## **JAVA QUESTIONS**

1. *What is Java Virtual Machine? Explain with Components of JVM Architecture?*
2. *Explain Polymorphism? How polymorphism Achieve in Java?*  
*Example: DMD (Dynamic Method Dispatching and function overloading in inheritance)*
3. *What is Abstraction? Explain with example of interface?*
4. *How Packages Work in Java? Explain with Access Specifier? (Page 258 part 1)*
5. *Explain Life Cycle of Applet with Applet Security? (Page 71 part 2)*
6. *Explain Event Class Hierarchies (Delegation Model)? Differentiate interface events and Adapter Classes. (Any AWT Example with event handling)*
7. *Explain Exception Handling in Java with Example of Multiple catch block?*
8. *Difference between Throw & Throws with Example? Can you own Exception give Example?*
9. *Explain Exception Propagation (Stack base) with Example? (page 2 part 2)*
10. *What is Multithread in Java with Example? Explain With its Life Cycle?*
11. *What is Synchronization how it helps in Locking? Explain with producer consumer Example?*
12. *Explain Daemon Threads with Example?*
13. *What is Streams in Java? Explain Character Stream and Byte Streams with Example?*
14. *What is Serialization and deserialization in Streams explain with Example?*
15. *What is JDBC? Explain its architecture with Connectivity Model?*
16. *Explain JDBC package? With Example?*
17. *What is java RMI? Explain its architecture with Example?*
18. *Difference between? TCP Socket and Datagram Socket(udp)*
19. *What is Sockets and Server Socket? Explain with Example?*
20. *What is UDP Socket? Explain with Example?*

### Useful Terms

- *Garbage Collection with Example System.gc() & finalize(). (page 9 part 1)*
- *Super Keyword (With Example of Method Overriding & Constructors in inheritance)*
- *Final Key Word*
- *this Object*
- *Difference Method Overriding & Method Overloading (Page 202 part 1)*
- *Difference class and Interface (page 243 part 1)*
- *Container (form, applet, canvas) Vs Component (Label, checkbox etc) (page 76 part 2)*
- *Layouts (113 part 2)*
- *Event Delegation Model*