

For B.E. B.C.A. M.C.A. M.C.M. B.S.C. Polytechnic



Programming

by Choudhari Sir



CCIT

Keeping Pace with Technology

An ISO 9001 : 2008 Certified Company

website: www.ccitindia.com

Rajapeth: 0721-2563615 GadgeNagar: 0721-2552289

Index

| Topic | Page |
|--|------|
| Java Basic : | |
| History, Features, JRE, JVM, ByteCode, JDK, Errors, JavaProgram, DataTypes, Operators, Identifiers , Literals, Type Casting, Variables types/Scope | 2 |
| Control Structures | |
| If statement ,Nested if , Ladder, Ternary Operator, while , for , do-while, switch , Nested Loops | 14 |
| Program Input | |
| Naming Conventions, Reading Input | 49 |
| Functions | |
| Functions basic, types of functions, method signature, method prototype, Recursion | 57 |
| OOPS | |
| Oops basic, classes, objects,data members , methods, method overloading, access specifiers, | 61 |
| Constructors | |
| Default constructor, Constructor with Args, Constructor Overloading, Initialization blocks, private constructors. | 81 |
| Static Members | |
| Static data members, static member functions , static initialization blocks, | 87 |
| Wrapper Classes | |
| Basics, class Integer, class Character | 91 |
| Arrays | |
| Single Dimensional arrays, MultiDimensional Arrays, Array of Objects, class Arrays , Command Line Arguments, Variable Arguments | 96 |
| Inheritance | |
| Basics, Method Overriding, super keyword, this keyword, types of inheritance, abstract keyword, Aggeration, final keyword, strictfp keyword, | 108 |

| | |
|--|-----|
| Polymorphism Basics, compiletime vs runtime polymorphism, instance of operator | 124 |
| Interfaces Basics, classes vs Interfaces, Abstract classes vs Interfaces, Interface Need, Default methods, | 128 |
| Inner Classes Basics , Anonymous classes, static nested classes | 135 |
| Packages Basics, Advantages, import statement, static import | 142 |
| Library Classes String, Math, String buffer , Vector, ArrayList, Set, HashSet, TreeSet, Map, Hashmap, Date, Calender, | 146 |
| Applets Baiscs, Applications Vs Applets, Life Cycle of Applet, Applet Tag, Applet Execution, Applet Parameters, Class Applet | 180 |
| AWT Graphics, Event Delegation Model, Button, TextField, TextArea, Checkbox, Label, List, Choice, Layout Managers, Font, Color, ScrollBar, Panel, Frame, Menu, Dialog, Mouse Events, Window Events, FileDialog, MediaTracker, Adapaters | 188 |
| Exceptions Basics, Uncaught Exceptions, Checked and Unchecked Exception, Exception Hierarchy, Exception Methods, try,catch, throw, throws, finally, Exception classes, User defined Exceptions, Nested try blocks, Exception Propogation | 229 |
| Input-Output / Streams Basics, File , InputStream, OutputStream, FileInputStream, FileOutputStream, Reader, FileReader, Writer, FileWriter, PrintStream, DataInputStream, DataOutputStream, Serialization, Deserialization | 246 |
| Multi-Threading Basics, Life Cycle of Thread, class Thread, Thread Priorities, Thread Synchronization, Thread Deadlock | 260 |

Java

Java is a simple object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi threaded and dynamic language, use to built internal application.

History of Java

James Gosling, the father of Java, was intent on building a low-cost, hardware-independent software platform using C++. James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects.



Interesting facts about Java-

- Java's old name was OAK (because of the Oak tree growing outside developer's house).
- Coffee Mug was deployed as Java's symbol, because the developers of Java drank and loved coffee.
- It was originally developed by Sun Microsystems, but later bought by Oracle.
- First major JDK (Java Development Kit) 1.0 was released on January 21, 1996.
- Android, one of the most famous mobile OS, is based on Java.(Android SDK also uses Java)
- Java has no concept of pointers unlike its predecessors.

Features

- **Simple.** Java's developers deliberately left out many of the unnecessary features of other high-level programming languages. For example, Java does not support pointer math, implicit type casting, structures or unions, operator overloading, templates, header files, or multiple inheritance.
- **Object-oriented.** Java uses classes to organize code into logical modules. At runtime, a program creates objects from the classes.
- **Statically typed.** All objects used in a program must be declared before they are used. This enables the Java compiler to locate and report type conflicts.
- **Compiled.** Before you can run a program written in the Java language, the program must be compiled by the Java compiler. The compilation results in a "**byte-code**" file that, while similar to a machine-code file, can be executed under any operating system that has a Java interpreter. This interpreter reads in the byte-code file and translates the byte-code commands into machine-language commands that can be directly executed by the machine that's running the Java program. You could say, then, that Java is both a compiled and interpreted language.
- **Multi-threaded.** Java programs can contain multiple threads of execution, which enables programs to handle several tasks concurrently. For example, a multi-threaded program can render an image on the screen in one thread while continuing to accept keyboard input from the user in the main thread.

- **Garbage collected.** Java programs do their own garbage collection, which means that programs are not required to delete objects that they allocate in memory. This relieves programmers of virtually all memory-management problems.
- **Robust.** Because the Java interpreter checks all system access performed within a program, Java programs cannot crash the system. Instead, when a serious error is discovered, Java programs create an exception. This exception can be captured and managed by the program without any risk of bringing down the system.
- **Secure.** The Java system not only verifies all memory access but also ensures that no viruses are hitching a ride with a running applet. Because pointers are not supported by the Java language, programs cannot gain access to areas of the system for which they have no authorization.
- **Extensible.** Java programs support native methods, which are functions written in another language, usually C++. Support for native methods enables programmers to write functions that may execute faster than the equivalent functions written in Java. Native methods are dynamically linked to the Java program; that is, they are associated with the program at runtime. As the Java language is further refined for speed, native methods will probably be unnecessary.
- **Well-understood.** The Java language is based upon technology that's been developed over many years. For this reason, Java can be quickly and easily understood by anyone with experience with modern programming languages such as C++.

Java bytecode

Java Byte Code is the language to which Java source is compiled and the Java Virtual Machine understands. Unlike compiled languages that have to be specifically compiled for each different type of computers, a Java program only needs to be converted to byte code once, after which it can run on any platform for which a Java Virtual Machine exists.

Java Virtual Machine

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

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

The JVM performs following main tasks:

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

JRE

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems/Oracle .

Different editions of Java platform-

J2SE(Java Platform, Standard Edition)

- Also known as Core Java, this is the most basic and standard version of Java. It's the purest form of Java, a basic foundation for all other editions.
- It consists of a wide variety of general purpose APIs (like java.lang, java.util) as well as many special purpose APIs
- J2SE is mainly used to create applications for Desktop environment.
- It consists all the basics of Java the language, variables, primitive data types, Arrays, Streams, Strings Java Database Connectivity(JDBC) and much more. This is the standard, from which all other editions came out, according to the needs of the time.

J2ME(Java Platform, Micro Edition)

- This version of Java is mainly concentrated for the applications running on embedded systems, mobiles and small devices.(which was a constraint before its development)
- Constraints included limited processing power, battery limitation, small display etc.

J2EE(Java Platform, Enterprise Edition)

- The Enterprise version of Java has a much larger usage of Java, like development of web services, networking, server side scripting and other various web based applications.
- J2EE is a community driven edition, i.e. there is a lot of continuous contributions from industry experts, Java developers and other open source organizations.
- J2EE uses many components of J2SE, as well as, has many new features of its own like Servlets, JavaBeans,
- J2EE uses HTML, CSS, JavaScript etc., so as to create web pages and web services. It's also one of the most widely accepted web development standard.

Software Engineering and Software Life Cycle

There are five major phases in the software life cycle.

Analysis phase

In the *analysis* phase, we perform a feasibility study. We analyze the problem and determine whether a solution is possible. Provided that a solution is possible, the result of this phase is a *requirements specification* that describes the features of a program. The features must be stated in a manner that are testable.

Design phase

In the *design* phase, we turn a requirements specification into a detailed design of the program. For an object-oriented design, the output from this phase will be a set of classes/objects that fulfill the requirements. The classes/objects must be fully defined, showing how they behave and how they communicate among themselves.

Coding phase

In the *coding* phase, we implement the design into an actual program, in our case, a Java program. Once we have a well-constructed design, implementing it into actual code is really not that difficult.

Testing phase

When the implementation is completed, we move to the *testing* phase. In this phase, we run the program using different sets of data to verify that the program runs according to the specification. Two types of testing are possible for object-oriented programs: *unit testing and integration testing*. With unit testing, we test classes individually. With integration testing, we test that the classes work together correctly. Activity to eliminate programming error is called *debugging*.

Operation phase

operation phase, in which the program will be put into actual use. The most important and time-consuming activity during the operation phase is *software maintenance*. After the software is put into use, we almost always have to make changes to it. Software maintenance means making changes to software. It is estimated that close to 70 percent of the cost of software is related to software maintenance.

Errors

The three kinds of errors you can encounter:

Compile Errors

Compile errors result from incorrectly constructed code. If you incorrectly type a keyword, omit some necessary punctuation, or use a do statement without a corresponding while statement at design time, Java detects these errors when you compile the application.

Run-Time Errors

Run-time errors occur while the application is running when a statement attempts an operation that is impossible to carry out. An example of this is division by zero. Suppose you have this statement:

Speed = Miles / Hours

If the variable Hours contains zero, the division is an invalid operation, even though the statement itself is syntactically correct. The application must run before it can detect this error.

Logic Errors

Logic errors occur when an application doesn't perform the way it was intended. An application can have syntactically valid code, run without performing any invalid operations, and yet produce incorrect results. Only by testing the application and analyzing results can you verify that the application is performing correctly.

General format of JAVA program

```
import statements
global declaration
class name
{
    public static void main (String arg [])
    {
        statements
        -----
    }
}
```

1) **Import Statement: -**

Java provide us different library classes. These classes are different packages, so, if we want to use them we have to import them by using import statement.

Eg: - import java.awt. *;
import java.awt.button;

2) **Global declaration: -**

In this section we can define classes & interface (user define data type) we can not define global variable & function like C & C++.

3) **Function main:-**

This is the entry point of our program execution of the program being with function main.

Java is complete object – oriented language. So even function main must be defined with in class.

a) void: - Indicates that this function main doesn't return any value.

b) static: - Indicates that this function can be called without creating object of that class.

c) public: - Indicates that the function is accessible.

d) String args[]: - It is an array of string object containing command line argument.

Output statements: -

1) *System.out.println (value):* Will print the value & new line character.

2) *System.out.print (value):* Will print the value on console.

3) *System.out.printf ("format String",arg1,...):* Will print the formatted String on console

For eg:

```
class test
{
    public static void main (string arg [ ])
    {
        System.out.println ("Welcome to JAVA");
    }
}
```

JDK (Java Development Kit): -

Oracle provides us this kit which contains different tools by using this tools we can perform different operation such as compilation, execution, documentation, debugging & so on. The JDK includes a private JVM and a few other resources to finish the development of a Java Application

1) Javac: - It is the java compiler which is used to convert our source code into byte code.

Syntax: - ***java filename***

It creates class file for all the classes present in our program.

2) Java: - It is the interpreter which is use executes our java program.

Syntax: - ***java classname***

It loads that class & calls its function main without creating object of that class.

3) AppletViewer :- AppletViewer is a standalone command-line program from Sun to run Java applets. Appletviewer is generally used by developers for testing their applets before deploying them to a website.

Syntax: - ***appletviewer filename***

4) jar :- combines multiple files into a single JAR archive file.

Syntax:- ***jar options jarfilename filelist***

5) JDB :- Java Debugger, commonly known as jdb, is a useful tool to detect bugs in Java programs.

Syntax :- ***jdb [options] [class] [arguments]***

EDIT-COMPILE-RUN-CYCLE

EDIT:

A java source program can be typed in any text editor and must be saved in a file with extension . java.

COMPILE:

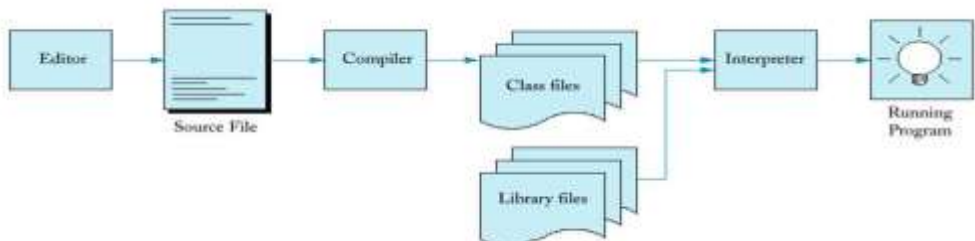
The source program can be compiled by using the java compiler i.e. javac. The compiler converts the source code into byte code.

Syntax: ***`javac SourceFileName`***

It will create class files for all the classes that are present in the program.

RUN: The byte code can be executed by using the Interpreter i.e. java . The Interpreter load the class and call its method main without creating object of the class.

Syntax: ***`java ClassName`***



Data types

- | | | | |
|----|---------|---------|-----------------------------|
| 1. | int | 4 bytes | |
| 2. | long | 8 bytes | <i>whole nos.</i> |
| 3. | short | 2 bytes | |
| 4. | float | 4 bytes | <i>floating point nos.</i> |
| 5. | double | 8 bytes | |
| 6. | char | 2 bytes | <i>(uni code char set)</i> |
| 7. | byte | 1 byte | <i>(single raw bytea)</i> |
| 8. | boolean | | <i>(True or False)</i> |

Operators

- 1) **Arithmetic:** - +, -, *, /, %, ++, --.
- 2) **Assignment:** - =, +=, -=, *=, /=, %=.
- 3) **Relational:** - <, >, <=, >=, ==, !=.
- 4) **Logical:** - &&, ||,
- 5) **Concat:** - + (to join two string & value)
 Eg:-
 "Ram" + "Seeta"
 "Ram" + 4117

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

Operator Associativity

the **associativity** (or fixity) of an **operator** is a property that determines how **operators** of the same precedence are grouped in the absence of parentheses.

Type Promotion

If an expression contains mixed type of value then result will be calculated according to higher datatype of value present in expression.

Java has rules for doing type promotion that are :

1. double
2. float
3. long
4. int
5. char or short
6. byte

Precedence & Associativity

| Category | Operator | Associativity |
|----------------|-----------------------------------|---------------|
| Postfix | () [] . (dot operator) | Left to right |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %= >>= <<= &= ^= = | Right to left |

Eg. class test
 { public static void main (String arg[])
 { int a=5, b=7, c;
 c = a + b;
 System.out.println ("Sum is " + c);
 }
 }

WAP given three no. to find mean.

```
class test
{ public static void main (String arg [ ])
  { int a=5, b=7, c=6;
    double m = ( a + b + c ) / 3;
    System.out.println ("Mean is " + m);
  }
}
```

WAP to Exchange values of 2 variables

```
class test
{public static void main (String arg [ ])
  {int a=5,b=8,c;
   c=a;
   a=b;
   b=c;
   System.out.println ("Values are " + a+ " and "+ b);
  }
}
```

WAP to find area & circumference from given radius

```
class test
{ public static void main (String arg [ ])
  {int r = 5;
   double a, c;
   a = 3.14 * r * r;
   c = 2 * 3.14 * r;
   System.out.println ("Area is " + a);
   System.out.println ("Circumference is " + c);
  }
}
```

WAP to find last digit of a given no.

```

class test
{
    public static void main (String arg [ ])
    {
        int n = 123;
        int l = n % 10;
        System.out.println ("Last digit is " + l);
    }
}

```

WAP to find sum of 4 digit no.

```

class test
{
    public static void main (String arg [ ])
    {
        int n = 1234;
        int a, b, c, d, s;
        a=n%10;
        b=n%100/10;
        c=n%1000/10;
        d=n/1000;
        s=a+b+c+d;
        System.out.println (" sum is "+s);
    }
}

```

WAP to find reverse of given 4 digit no.

```

class test
{
    public static void main (String arg [ ])
    {
        int n = 1234;
        int a, b, c, d, r;
        a=n%10;
        b=n%100/10;
        c=n%1000/10;
        d=n/1000;
        r=a*1000+b*100+c*10+d;
        System.out.println ("Reverse No is " + r);
    }
}

```

Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called identifiers.

In java there are several points to remember about identifiers. They are as follows:

- All identifiers should begin with a letter (A to Z or a to z), currency character (\$) or an underscore (_).
- After the first character identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly identifiers are case sensitive.
- Examples of legal identifiers: age, \$salary, _value, __1_value
- Examples of illegal identifiers : 123abc, -salary

Java Literals

Java literals are fixed or constant values in a program's source code.

Java Integer Literals

Integer literals in Java can be used in three flavors in a Java program those are *decimal*, *octal* (base 8), and *hexadecimal* (base 16).

```
int octLit = 0400; //octal equivalent of decimal 256
int hexLit = 0x100; //hexadecimal equivalent of decimal 256
int decLit = 256; // decimal 256
```

Java Floating-point Literals

Floating-point literals in Java default to double precision. To specify a float literal, you must append an `f` or `F` to the constant.

```
float ff = 89.0f; //OK
double dou = 89.0D; //OK
double doub = 89.0d; //OK
double doubl = 89.0; //OK, by default floating point literal is double
```

Java Character Literals

Character literals are enclosed in single quotes when they are used for example, 'a' or 'A'. Whereas string literals are enclosed in double quotes for example, "Coffea arabica".

```
char charLit = 'a';
String strLit = "String Literal";
```

Type Casting

Assigning a value of one type to a variable of another type is known as **Type Casting**. Java data type casting comes with 3 flavors.

Implicit casting :

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

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

Explicit casting (narrowing conversion)

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

```
float a=3.2;
int b=(int )a; //Explicit casting by using cast operator
```

3. Boolean casting

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

Scope / Types of Variables

A variable's scope is the region of a program within which the variable can be referred to by its simple name.

The location of the variable declaration within your program establishes its scope and places it into one of these three categories:

- Local variables
- Instance variables
- Class/static variables

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . ***ObjectReference.VariableName***.

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name ***ClassName.VariableName***.

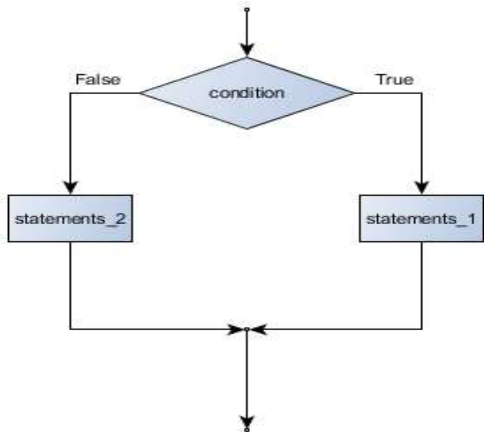
If Statement

It is used to conditionally execute a block of code.

Syntax:-

```

if(condition)
{
    statements
}
else
{
    Statements
}
optional
}
next statement
  
```



if condition is true then

statements within if block are executed

if condition is false then statements within else block are executed

note: else block is optional.

Condition can be specified by using a Boolean exp. A Boolean exp. Can be created by using Relational & Logical Operators.

WAP to check if a given no is an even no. of odd no.

class test

```

{public static void main (String arg [ ])
{int n = 7;
    if (n%2==0)
        System.out.println ("No. is even");
    else
        System.out.println ("No. is even");
    }
}
  
```

WAP to check if a triangle can be formed from given three angle.

class test

```
{ public static void main (String arg [])
    { int a = 57 , b = 48, c = 93, z;
      if (a + b + c == 180 )
          System.out.println ("triangle can be formed");
      else
          System.out.println ("triangle can not be formed");
    }
}
```

WAP to print total marks of a student from given marks of 5 subject, also print percentage if student if pass. (Passing marks 40 per sub.)

class marks

```
{public static void main (String arg [ ])
    {
        int a = 57 , b = 48, c = 93, d = 80, e = 84, total;
        double per;
        total = a + b + c + d + e;
        System.out.println ("Total marks = " + total);
        if (a >= 40 && b >= 40 && c >= 40 && d >= 40 && e >= 40)
        {
            per = total / 5.0;
            System.out.println ("Student is pass");
            System.out.println ("Percentage is " + per);
        }
        else
            System.out.println ("Student is fail");
    }
}
```

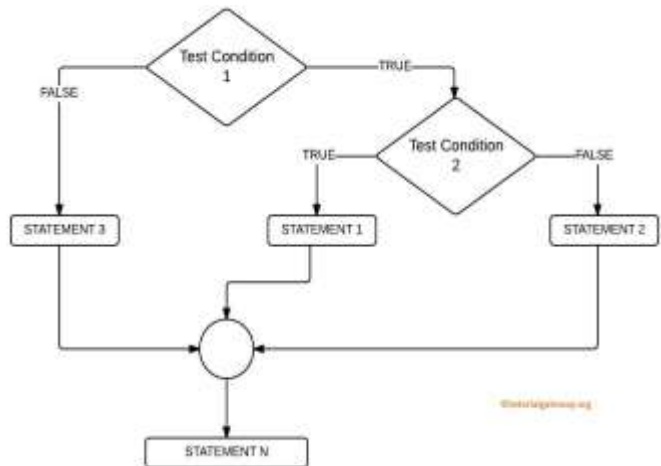
Nested if

If a **if** statement is used within an if statement then such a control structure is called as nested if.

Syntax: -

```
if (condition1)
{
  if (condition2)
  {
    Statements
  }
}
```

WAP to print total marks of a student from given marks of 5 subject, also print percentage if student if pass. (Passing marks 40 per sub.)



```

class marks
{
    public static void main (String arg [ ])
    {
        int a = 57 , b = 48, c = 93, d = 80, e = 84, total;
        double per;
        total = a + b + c + d + e;
        per = total / 5.0;
        System.out.println ("Total marks = " + total);
        if (a >= 40 && b >= 40 && c >= 40 && d >= 40 && e >= 40)
        {
            if (per >= 60)
                System.out.println ("Student get Ist class");
        }
    }
}
  
```

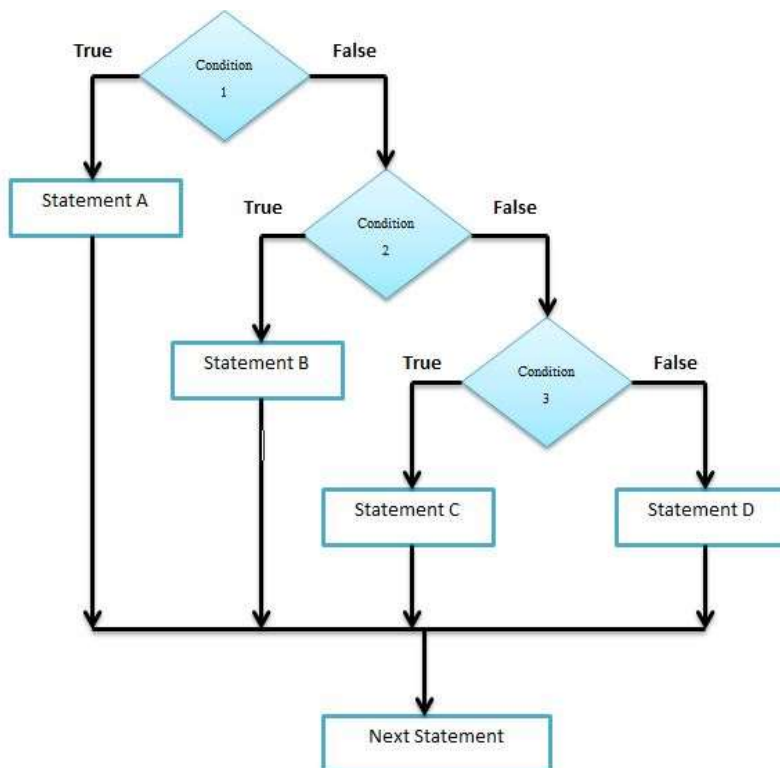
Ladder Structure

If a **if** statement is used within an **if** statement then such a control structure is called ladder structure.

Syntax: - if (condition)

else if (condition)

else if (condition)



WAP to print division according to given percentage.

| Per | div. |
|------------|-------------------|
| ≥ 75 | DT |
| 60-75 | I st |
| 50-60 | II nd |
| 40-50 | III rd |
| < 40 | fail |

class marks

```
{
    public static void main (String arg [])
    {
        double per = 57.37;
        if (per >= 75)
            System.out.println ("Distinctions");
        else if (per >= 60)
            System.out.println ("Ist division");
        else if (per >= 50)
            System.out.println ("IInd division");
        else if (per >= 40)
            System.out.println ("IIIrd division");
        else if (per >= 35)
            System.out.println ("Pass");
        else
            System.out.println ("Fail");
    }
}
```

WAP to find greatest no. from given three different no.

class test

```
{
    public static void main ( String arg [ ] )
    {
        int a = 69, b = 66, c = 97;
        if (a > b && a > c)
            System.out.println( "a is greater");
        else if (b > a && b > c)
            System.out.println( "b is greater");
        else
            System.out.println( "c is greater");
    }
}
```

Conditional / ternary Operator (? :)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {  
  
    public static void main(String args[]){  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
  
        b = (a == 10) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
    }  
}
```

Loops

Loops are used to repeatedly execute a block of codes. Java provides us with different looping structures.

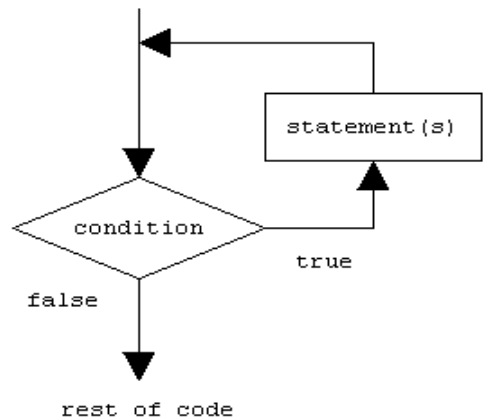
- 1> while loop
- 2> for loop
- 3> do-while loop

while loop

Statements within this loop are repeatedly executed while the condition is true. Program control is transferred to the next statement only when the condition becomes false.

Syntax: -

```
while (condition )
{
    Block of
statements;
}
Next statements;
```



WAP to print all no. from 1 to 10.

```
class test
{
    public static void main (String args [ ])
    {
        int I=1;
        while ( I <= 10)
        {
            System.out.println (I);
            I++;
        }
    }
}
```


}

WAP to print all no. from 10 to 1.

```

class test
{
    public static void main (String args [ ])
    {
        int I=10;
        while ( I >= 1)
        {
            System.out.println (I);
            I--;
        }
    }
}

```

WAP to print all no. from 100 to 1. Which are divisible by 2 OR 5

```

class test
{
    public static void main (String args [ ])
    {
        int I=100;
        while ( I >= 1)
        {
            if(I %2 == 0 || I %5 == 0)
                System.out.println (I);
            I--;
        }
    }
}

```

WAP to print all no. from 1 to n. Which divides the no. n perfectly.

```

class test
{
    public static void main (String args [ ])
    {
        int I=1,n=27;
        while ( I <= n)
        {
            if(n %I == 0 )
                System.out.println (I);
            I++;
        }
    }
}

```

for loop

- Statement within this loop are repeatedly executed while condition is true.

Syntax: -

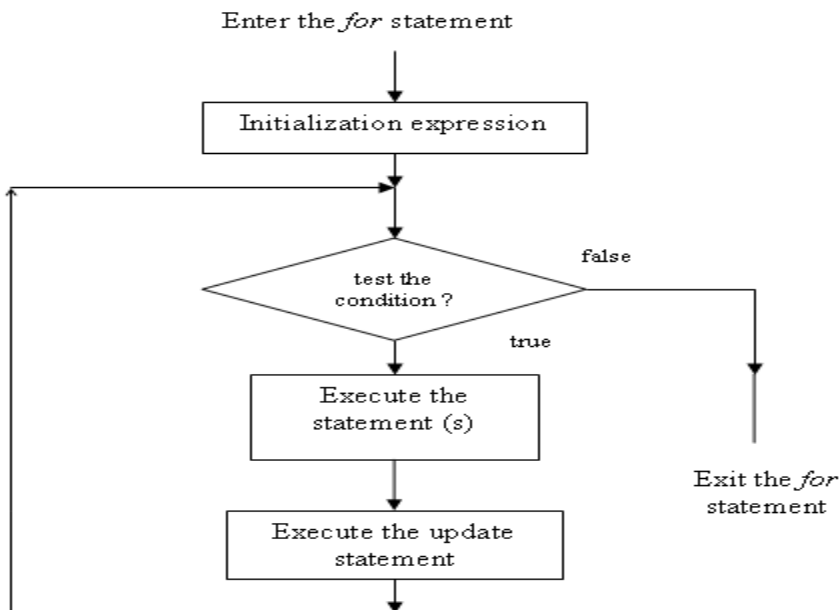
for(initialization, condition, evaluation)

{
Statement

}

next statements

Specialty of this loop is that initialization, condition, evaluation is specified at the starting of loop.



WAP to print all no. from 1 to 10.

```
class test
{
    public static void main (String args [])
    {
        int I;
        for (I = 1; I <= 10 ; I++)
            System.out.println (I);
    }
}
```

WAP to print all no. from 1 to n. Which divides the no. n perfectly.

```
class test
{
    public static void main (String args [ ])
    {
        int I=1,n=27;
        for(int I=1;I<=n;I++)
        {
            if(n %I == 0 )
                System.out.println (I);
        }
    }
}
```

For-each loop

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

Advantage:

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Syntax

```
for(data_type variable : array | collection)  
{  
Statements...  
.....  
}
```

For eg:

```
class ForEachExample1  
{  
    public static void main(String args[])  
    {  
        int arr[]={12,13,14,44};  
  
        for(int i:arr)  
        {  
            System.out.println(i);  
        }  
    }  
}
```

Nested Loops

If a loop is used within a loop then such a control structure is called as Nested Loop. Nested loops are used to repeatedly execute a block of code which is itself repeating a block of code.

WAP to print following o/p.

```

* * * * * 12345
* * * *   1234
* * *     123
* *       12
*         1

```

```

1]    class test
        {public static void main (String args [ ])
        { for(int a=0,b=5;b>=1;a++,b--)
            {
                for (int I = 1; I <= a ; I++)
                    System.out.println (" ");
                for (int I = 1; I <= b ; I++)
                    System.out.println ("* ");
            }
        }
    }

```

```

2]    class test
        {public static void main (String args [ ])
        { int I,N;
          for(N=5;N>=1;N--)
            {
                for (I = 1; I <= N ; I++)
                    System.out.print (I);
                System.out.println();
            }
        }
    }

```

do-while loop

Statements within this loop are repeatedly executed while condition is true specially of this loop is that, it is executed at least once even if condition at least once even if condition is false.

Syntax:-

do

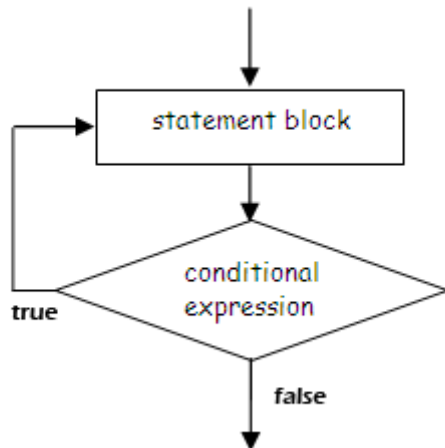
{ Statements;

.....

}

while (condition);

Next statements



WAP to read a no while the no. entered is not equals to 0 and find sum of all nos. entered.

class sum

{public static void main (String args [])

{int Sum=0;

do { String s=JOptionPane.showInputDialog (null, "Enter a no.");

int N=Integer.parseInt(s);

Sum=Sum + N;

}

while(N!=0);

System.out.println("Sum is =" + Sum);

}

}

}

Jump Statements

Java provides us different jump statements which are used to control flow of execution of program

1) break Statement:- This statement is used to throw program control out of the loop or a switch statement.

Eg:- class pc

```

    { public static void main (String args[])
      { int I =1;
        while (I <=10)
        {
          System.out.println(i);
          if ( I ==5 )
            break;
          I ++;
        }
      }
    }

```

2) continue statement: - This statement is used to throw program control at the starting of loop.

class pc

```

    { public static void main( String arg [] )
      { int I = 0;
        while (I <10)
        {I ++;
          if ( I %2 ==0 )
            continue;
          System.out.println (I);
        }
      }
    }

```


3) Labeled break And continue Statement

These statements are used to throw program control out of nested loop or to throw at the starting of nested loop.

Syntax:- break label;

OR continue label;

Label can be defined by using syntax identifiers or colon.

```
class pc
{
    public static void main( String args [] )
    {
        -----
        -----
        A : while (-----)
        {
            B : while (-----)
            {
                -----
                -----
                break A;
            }
            -----
            -----
        }
        -----
        -----
    }
}
```

switch Statement

This statement is used to check a variable for different value & a/c to different value of variable different cases are executed. Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths.

Syntax:-

switch(variable)

{

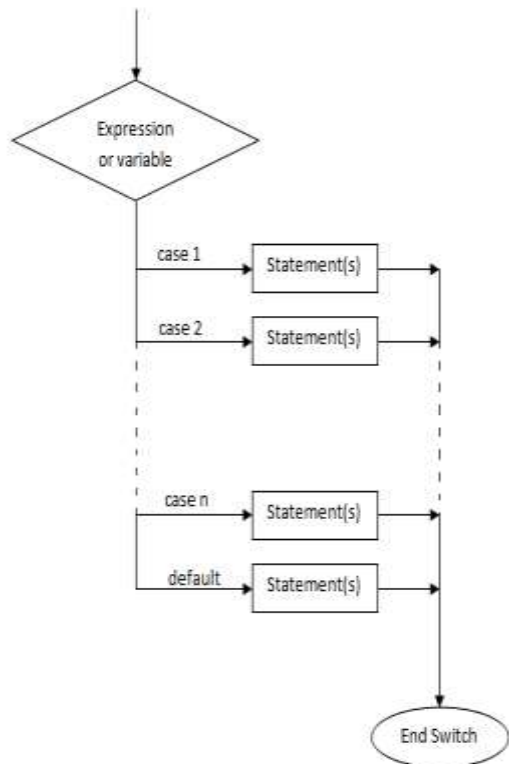
case value: -----

case value: -----

....

default: -----

}



- * If no matching case is found then code in the default block is executed.
- * default block is optional.
- * A break statement can be used to throw program control out of the switch otherwise be executed without checking their value.
- * A switch works with the byte, short, char, and int primitive data types. It also works with enumerated type and a few special classes that "wrap" certain primitive types: Character, Byte, Short, and Integer

WAP to read a single digit no & print that no. in word

```
class digit
```

```
    { public static void main (String arg[] )
      { String s = JOptionPane.showInputDialog (null, " Enter a single digit no");
        int N = Integer.parseInt (s);
        switch( N )
        {
          case 0: System.out.println ("Zero");
                  break;
          case 1: System.out.println ("One");
                  break;
          case 2: System.out.println ("Two");
                  break;
          case 3: System.out.println ("Three");
                  break;
          case 4: System.out.println ("Four");
                  break;
          case 5: System.out.println ("Five");
                  break;
          case 6: System.out.println ("Six");
                  break;
          case 7: System.out.println ("Seven");
                  break;
          case 8: System.out.println ("Eight");
                  break;
          case 9: System.out.println ("Nine");
                  break;
          default :System.out.println ("Not a single digit no.");
        }
      }
    }
```

WAP to read a single digit no & print all the no. from that no. to 9 & on word.

```
class digit
{
    public static void main (String arg[] )
    {
        String s = JOptionPane.showInputDialog (null, "Enter a single digit no");
        int N = Integer.parseInt (s);
        switch( N )
        {
            case 0: System.out.println ("Zero");
            case 1: System.out.println ("One");
            case 2: System.out.println ("Two");
            case 3: System.out.println ("Three");
            case 4: System.out.println ("Four");
            case 5: System.out.println ("Five");
            case 6: System.out.println ("Six");
            case 7: System.out.println ("Seven");
            case 8: System.out.println ("Eight");
            case 9: System.out.println ("Nine");
            default: System.out.println ("Not a single digit no.");
        }
    }
}
```

Final Variables

You can declare a variable in any scope to be *final* . The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other programming languages. To declare a final variable, use the final keyword in the variable declaration before the type:

```
final int aFinalVar = 0;
```

The previous statement declares a final variable and initializes it, all at once. Subsequent attempts to assign a value to aFinalVar result in a compiler error. You may, if necessary, defer initialization of a final *local* variable. Simply declare the local variable and initialize it later, like this:

```
final int blankfinal;
. . .
blankfinal = 0;
```

Naming Conventions

Java provides us different library classes for naming these classes & their method they have followed some conventions.

| Name | Convention |
|----------------|--|
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc. |

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

Reading data

If a program requires some user input then such input can be provided in different ways.

1) By Using **showInputDialog** function:-

These functions show a small input window where we can enter our data.

Syntax:-

JOptionPane.showInputDialog (component parent,, string message)

This function return the entered value as a string

WAP to read a no & print all no. from 1to that no.

```
class test
{
    public static void main (String args [ ])
    {
        String s=JOptionPane.showInputDialog (null, "Enter a no.");
        int N=Integer.parseInt(s);
        for(int i=1; i<=N; i++)
            System.out.println(i);
    }
}
```

WAP to read a no & find sum of all no from 1 to that no.

```
class test
{
    public static void main (String args [])
    {
        int Sum=0;
        String s=JOptionPane.showInputDialog (null, "Enter a no.");
        int N=Integer.parseInt(s);
        for(int i=1; i<=N; i++)
            Sum=Sum + i;
        System.out.println( "Sum is =" + Sum);
    }
}
```

2> By using Command line Arguments

Whenever a program is run or executed from command prompt, we can pass some data (command line arguments). This data are send to the function main as arguments in a string array, where this data can be used to perform some operations.

```
class fact
{
    public static void main (String args [ ])
    {
        int f=1;
        int N=Integer.parseInt(args [0]);
        for (int i=1; i<=N; i++)
            f=f*i;
        System.out.println ( "Factorial is =" + f);
    }
}
```

3> By Using Scanner class:

Scanner class is widely used to parse text for string and primitive types using regular expression.

```
import java.util.Scanner;
class ScannerTest{
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter your rollno");
        int rollno=sc.nextInt();
        System.out.println("Enter your name");
        String name=sc.next();
        System.out.println("Enter your fee");
        double fee=sc.nextDouble();
        System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
        sc.close();
    }
}
```

4> By using System.in

The InputStream object System.in can be used to read data. But this object provides us methods to read only bytes. So we can wrap it in DataInputStream to read input data.

```
import java.io.*;
class fact
{
    public static void main (String args [ ])
    {
        int f=1;
        try    {
            DataInputStream din=new
            DataInputStream(System.in);
            System.out.print("Enter a No ");
            String s=din.readLine();
            int N=Integer.parseInt(s);
            for (int i=1; i<=N; i++)
            {
                f=f +i;
            }
            System.out.println ( "Factorial is =" + f);
        }
        catch(Exception er)
        {
            System.out.println(er);
        }
    }
}
```


Bitwise Operators

Java's *bitwise* operators operate on individual bits of integer values. If an operand is shorter than an int, it is promoted to int before doing the operations. It helps to know how integers are represented in binary. For example the decimal number 3 is represented as 11 in binary and the decimal number 5 is represented as 101 in binary. Negative integers are store in *two's complement* form. For example, -4 is 1111 1111 1111 1111 1111 1111 1100.

| Op | Name | Example | Result | Description |
|-------------------------------|-------------|---------------------------------|--------|--|
| <code>a & b</code> | and | <code>3 & 5</code> | 1 | 1 if both bits are 1. |
| <code>a b</code> | or | <code>3 5</code> | 7 | 1 if either bit is 1. |
| <code>a ^ b</code> | xor | <code>3 ^ 5</code> | 6 | 1 if both bits are different. |
| <code>~a</code> | not | <code>~3</code> | -4 | Inverts the bits. |
| <code>n << p</code> | left shift | <code>3 <<< 2</code> | 12 | Shifts the bits of n left p positions. Zero bits are shifted into the low-order positions. |
| <code>n >> p</code> | right shift | <code>5 >> 2</code> | 1 | Shifts the bits of n right p positions. If n is a 2's complement signed number, the sign bit is shifted into the high-order positions. |
| <code>n >>> p</code> | right shift | <code>-4 >>> 28</code> | 15 | Shifts the bits of n right p positions. Zeros are shifted into the high-order positions. |

```
public class Test {  
    public static void main(String args[]) {  
        int a = 60;    /* 60 = 0011 1100 */  
        int b = 13;    /* 13 = 0000 1101 */  
        int c = 0;  
  
        c = a & b;      /* 12 = 0000 1100 */  
        System.out.println("a & b = " + c );  
  
        c = a | b;      /* 61 = 0011 1101 */  
        System.out.println("a | b = " + c );  
  
        c = a ^ b;      /* 49 = 0011 0001 */  
        System.out.println("a ^ b = " + c );  
  
        c = ~a;         /* -61 = 1100 0011 */  
        System.out.println("~a = " + c );  
  
        c = a << 2;     /* 240 = 1111 0000 */  
        System.out.println("a << 2 = " + c );  
  
        c = a >> 2;     /* 15 = 1111 */  
        System.out.println("a >> 2 = " + c );  
  
        c = a >>> 2;    /* 15 = 0000 1111 */  
        System.out.println("a >>> 2 = " + c );  
    }  
}
```

Enum

Enum in java is a data type that contains fixed set of constants.

Java Enums can be thought of as classes that have fixed set of constants.

Points to remember for Java Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

```
class EnumExample
{
    public enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args)
    {
        for (Season s : Season.values())
            System.out.println(s);
    }
}
```

values() method

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

++ and -- operator as prefix and postfix

- The ++/-- operator increments/decrements its single operand by one.
- The behavior of increment operator during an assignment operation depends on its position relative to the operand whether it is used in prefix or postfix mode.
- When used in prefix mode, it increments the operand and evaluates to the incremented value of that operand. When used in postfix mode, it increments its operand, but evaluates to the value of that operand before it was incremented.

```
int x = 5, y;

// Demonstrating prefix increment
// first x will be incremented then
// updated value of x will be assigned to y
y = ++x;
System.out.println("y : " + y); //will print y : 6
System.out.println("x : " + x); //will print x : 6

// Demonstrating postfix increment
// first value of x will be assigned to y
// then x will be incremented
y = x++;
System.out.println("y : " + y); //will print y : 6
System.out.println("x : " + x); //will print x : 7

//If increment is made in an independent
//statement, prefix and postfix modes make no difference.
++x;
System.out.println("x : " + x); //will print x : 8

x++;
System.out.println("x : " + x); //will print x : 9
```

Functions

A function is block of code design to perform some task.

Functions are of two types.

1) **Functions returning value:-**

Such functions are called to perform some calculations (task) which returns us some value.

Eg:- `Z = Math.sqrt (9);`
`P=fact (5);`

Such functions return us some value, so they must always be used in an expression.

2) **Function not returning value:-**

Such functions are called just to perform some task. They doesn't return any value so they cant be used in an expression

Eg:- `g.drawline(10, 20, 50, 60);`

General format to define function:-

```
returntype functionname (datatype arg1, datatype arg2,...)
{
Statements;
-----
return value;
}
```

Return type:-

It indicates the datatype of the value which function is going to return. It can be any datatype such as int, long, float, double, byte, boolean, char , an object etc.

if function is not returning any value then return type must be void.

* **Design a function star which will print 50 stars.**

```
class pc
{
    public static void main( String arg[] )
    {
        System.out.println( "CCIT");
        star();
        System.out.println( "Amravati");
        star();
    }
    static void star()
    {
        for (int i=1; i<= 50; i++)
            System.out.println( "*" );
    }
}
```

Functions with arguments [parameterized functions]

While calling a function we can pass some data. This data is pass on to the functions as argument, where a/c to the argument value, the function can performed different task.

Design a function repeat which will repeatedly print a character specified no of time

```
class pc
{ public static void main ( String arg [] )
    {
        repeat ( '#',20);
        repeat ( '*',30);
        repeat ( '&',10);
    }
    static void repeat(char a, int n)
    {for (int i=1; i<= n; i++)
        System.out.println( a );
    }
}
```

Functions Returning Values

If you want to use function in an expression then such function must return a value, by using a return statement.

```
class pc
{
    public static void main ( String arg [] )
    {
        int z =100 + fact (5);
        System.out.println ( "Result is "+ z);
    }
    static int fact (int n)
    {
        int f =1;
        for (int i=1; i<= n; i++)
            f = f * i;
        return f;
    }
}
```

Method Signature

A method signature is part of the method declaration. It is the combination of the method name and the parameter list.

The reason for the emphasis on just the method name and parameter list is because of overloading. It's the ability to write methods that have the same name but accept different parameters. The Java compiler is able to discern the difference between the methods through their method signatures.

Method Prototype

It is the combination of the return type ,method name and the parameter list as well as function modifiers.

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
class fabo
{
    private static int index = 0;
    private static int stoppingPoint = 40;

    public static void main (String[] args)
    {
        int n1 = 0;
        int n2 = 1;
        System.out.println("index: " + index + " -> " + n1);
        fibonacciSequence(n1, n2);
    }

    public static void fibonacciSequence(int n1, int n2)
    {
        System.out.println("index: " + index + " -> " + n2);

        // make sure we have set an ending point so this Java recursion
        // doesn't go on forever.
        if (index == stoppingPoint)
            return;

        // make sure we increment our index so we make progress
        // toward the end.
        index++;

        fibonacciSequence(n2, n1+n2);
    }
}
```


OOPS

Java is an object oriented language. It provides us a program environment where we can create object & perform operations on them.

Java supports the following fundamental concepts:

| | |
|---|--|
| <ul style="list-style-type: none"> • Polymorphism • Inheritance • Encapsulation • Abstraction | <ul style="list-style-type: none"> • Classes • Objects • Instance • Method |
|---|--|

OOPs Vs Procedure-oriented programming

| | Procedure Oriented Programming | Object Oriented Programming |
|---------------------|---|---|
| Divided Into | In POP, program is divided into small parts called functions . | In OOP, program is divided into parts called objects . |
| Importance | In POP, Importance is not given to data but to functions as well as sequence of actions to be done. | In OOP, Importance is given to the data rather than procedures or functions because it works as a real world . |
| Approach | POP follows Top Down approach . | OOP follows Bottom Up approach . |

| | | |
|--------------------------|---|--|
| Access Specifiers | POP does not have any access specifier. | OOP has access specifiers named Public, Private, Protected, etc. |
| Data Moving | In POP, Data can move freely from function to function in the system. | In OOP, objects can move and communicate with each other through member functions. |
| Expansion | To add new data and function in POP is not so easy. | OOP provides an easy way to add new data and function. |
| Data Access | In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system. | In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data. |
| Data Hiding | POP does not have any proper way for hiding data so it is less secure . | OOP provides Data Hiding so provides more security . |
| Overloading | In POP, Overloading is not possible. | In OOP, overloading is possible in the form of Function Overloading and Operator Overloading. |
| Examples | Example of POP are : C, VB, FORTRAN, Pascal. | Example of OOP are : C++, JAVA, VB.NET, C#.NET. |

Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Inheritance

Inheritance is the ability, offered by many OOP languages, to derive a new class (the derived or inherited class) from another class (the base class). The derived class automatically inherits the properties and methods of the base class.

For example, you could define a generic Shape class with properties such as Color and Position and then use it as a base for more specific classes (for example, Rectangle, Circle, and so on) that inherit all those generic properties. You could then add specific members, such as Width and Height for the Rectangle class and Radius for the Circle class.

Message

A **message** is a request to an object to invoke one of its methods. A message therefore contains

- the **name** of the method and
- the **arguments** of the method.

This interacting between objects is based on *messages* which are sent from one object to another asking the recipient to apply a method on itself. we could create new objects and invoke methods on them. For example,

```
Account a=new Account(4117,5000)
/* Define a new Account object with account no 4117 & balance
5000 */
a.deposit(1500); /* deposit 1500 into account */
```

Object

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. **Object is an instance of a class.** Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user.

Class

A class is a generic definition of an object. It is a blue print of an object. The class definition describes all the properties, behavior, and identity of objects present within that class.

Definition: - "A class is uses define data type were we can bind, data & related function together".

syntax

```

class name
{
  Data members
  -----
  -----
  -----
  Member functions
  -----
  -----
  -----
}
```

A class can be defining in terms of its data members & member function.

Data members

They indicate information about an object current state of the object.

Syntax:-

[Access modifiers] datatype membername [= value] ;

Member function

They indicates the operation which we can perform on the object.

Syntax: **[Access modifier] datatype function name (arg list)**

```
{
Statements;
-----
-----
-----
return value;
}
```

Eg:-

```
class rectangle
{
    int length, breath;
    void area()
    {
        int a = length * breath;
        System.out.println( " Area is = "+ a);
    }
    void perimeter()
    {
        int p = 2 * (length * breath);
    }
}
```

Object : An object is an instance of a class. Objects can be created whenever required by using a keyword "new".

Reference : A reference is a variable in which we can store the id of an object.

Syntax:- classname referencelist;

Eg:- rectangle a , b , c;

Creating objects :

Syntax:- reference = new classname ();

It will create an object and will return the id of an object, which we can store in a reference variable. Whenever an object is created the space is reserved for its data members..

Note:

1. We can assign the value of a reference variable to another reference variable.
2. Reference Variable is used to store the address of the variable.
3. Assigning Reference will not create distinct copies of Objects.
4. All reference variables are referring to the same Object.

There are different ways to create objects in java:

Using new keyword:

This is the most common way to create an object in java. Almost 99% of objects are created in this way.

```
MyObject object=new Object();
```

Using Class.forName():

If we know the name of the class & if it has a public default constructor we can create an object in this way.

Syntax:

```
MyObject obj=(MyObject)class.forName("object").newInstance();
```

Using clone():

The clone() can be used to create a copy of an existing object.

Syntax:

```
MyObject obj=new MyObject();
```

```
MyObject object=(MyObject )obj.clone();
```

Using Object Deserialization:

Object deserialization is nothing but creating an object from its serialized form.

Syntax:

```
objectInputStream istream=new objectInputStream(some data);
```

```
MyObject object=(MyObject) instream.readObject();
```

Using newInstance() method

```
Object obj = DemoClass.class.getClassLoader().loadClass("DemoClass").newInstance();
```


Objects Vs Classes

| Object | Class |
|---|--|
| Object is an instance of a class. | Class is a blueprint or template from which objects are created. |
| Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a group of similar objects . |
| Object is a physical entity. | Class is a logical entity. |
| Object is created through new keyword mainly e.g. Student s1=new Student(); | Class is declared using class keyword e.g. class Student{} |
| Object is created many times as per requirement. | Class is declared once . |
| Object allocates memory when it is created . | Class doesn't allocated memory when it is created . |
| There are many ways to create object in java such as new keyword, newInstance() method, clone() etc. | There is only one way to define class in java using class keyword. |

Accessing Members:-

Members of an object can be access by using syntax.

recferecneName.memberName

Access specifiers

They indicate accessibility of member of a class. It can be private, public, protected or default (not specified).

private members:-Are accessible only within the class i.e. only by the members of that class.

Generally data members are kept private because we want to hide internal details of the objects.

public members:-Are accessible in the class as well as outside the class.

Generally member functions are kept public.

protected members:- Can be access in the class as well as in all the class derived from that class.

Default (not specified) :- If access modifier is not specifies then they are of type default or friendly and are accessible within the package.

eg: design a class rectangle containing datamembers length ,breath & member functions area & perimeter.

```
class rectangle
{
    int length, breath;
    void area()
    { int a = length * breath;
      System.out.println( " Area is = "+ a);
    }
    void perimeter()
    { int p = 2 * (length * breath);
    }
}
class pc
{public static void main ( String arg [] )
  { Rectangle a, b;
    a=new Rectangle ();
    b=new Rectangle ();
    a.length = 3;
    a.breadth = 4;
    b.length = 5;
    b.breadth = 4;
    b.perimeter ();
    a.area ();
    b.area ();
  }
}
```

Data member initialization

- 1) Numeric data members are by defaults to zero.
- 2) Boolean data members are by defaults false.
- 3) Object reference are initialize to null

Setter functions

These functions are used to set values of private data members. Generally their name starts with the name 'set'.

Eg:-

```
class Rectangle
{ private int length, breadth;
  public void area ()
    {int a = length * breadth;
     System.out.println (" Area  is = " + a);
    }
  public void perimeter ()
    {int p = 2 * ( length + breadth );
     System.out.println ("Perimeter is =" + p);
    }
  public void setdimension (int l, int b)
    {length = l;
     breadth = b;
    }
}

class pc
{public static void main ( String args [] )
  { Rectangle a, b;
    a = new Rectangle ();
    b = new Rectangle ();
    a.setdimention (3,7);
    b.setdimention (4,4);
    a.area ();
    b.perimeter ();
    b.area ();
  }
}
```

Design a class box containing data member l, b, h & member function setdimention & volume.

```
class Box
{ private int l, b, h;
  public void setdimention (int x, int y, int z)
    { l = x;
      b = y;
      h = z;
    }
  public void volume ()
    { int v = l * b * h;
      System.out.println ("Volume is =" + v);
    }
}
```

```
class pc
{
  public static void main ( String args [] )
  {
    Box a, b;
    a = new Box ();
    b = new Box ();
    a.setdimention (3, 7, 9);
    b.setdimention (4, 4, 4);
    a.volume ();
    b.volume ();
  }
}
```

Design a class worker containing data member wages & wdays & member function setdata & payment.

```
class Worker
{
    private int wages, wdays;
    public void setdata (int x, int y)
    {
        wages = x;
        wdays = y;
    }
    public void payment ()
    {
        int p = wages * wdays;
        System.out.println ("Payment is =" + p);
    }
}

class pc
{
    public static void main (String args [])
    {
        Worker a, b;
        a = new Worker ();
        b = new Worker ();
        a.setdata (250, 23);
        b.setdata (210, 44);
        a.payment ();
        b.payment ();
    }
}
```

Method overloading

Defining multiple methods in a class having same name is called as method overloading. Only precaution to be take is that no. of arguments & type of arguments must be different. Compiler decides which method is called depending on no. of arguments & type or type of argument that are pass while calling the method. By defining multiple methods we can call that method in multiple ways.

Rules for Method Overloading

1. Overloading can take place in the same class or in its sub-class.
2. Constructor in Java can be overloaded
3. Overloaded methods must have a different argument list.
4. Overloaded method should always be the part of the same class (can also take place in sub class), with same name but different parameters.
5. The parameters may differ in their type or number, or in both.
6. They may have the same or different return types.
7. It is also known as compile time polymorphism.

Design a class box whose dimension can be set by passing 3 arguments or by passing single argument.

```

class box
{ private int l, b, h;
  public void setdimension (int x, int y, int z)
    {l = x;    b = y;    h = z;
    }
  public void setdimension (int x )
    {l = b = h = x;
    }
  public void volume ()
    { int v = l * b * h;
      System.out.println ("Volume is"+ v);
    }
}

class pc
{public static void main (String args [ ])
  { box a, b;
    a = new box ();
    b = new box ();
    a.setdimention (4, 5, 6);
    b.setdimention (5);
    a.volume ();
    b.volume ();
  }
}

```


Method Returning value

If we want to use our method in an expression then such method must return a value using a return statement.

Ex:-

```
class worker
{
    private int wages, wdays;
    public void setdata (int x, int y)
    {
        wages = x;
        wdays = y;
    }
    public int payment ()
    {
        int p = wages * wdays;
        return p;
    }
}

class pc
{
    public static void main (String args [ ])
    {
        worker a, b;
        a = new worker ();
        b = new worker ();
        a.setdata (250, 23);
        b.setdata (210, 20);
        int n = a.payment () + b.payment ();
        System.out.println ("Payment is = "+ n);
    }
}
```

Constructors

Are special member functions of a class which gets automatically invoke when ever an object of its class is created. They are use for initialization of Object.

Rules/Properties/Characteristics of a constructor:

1. Constructor name must be similar to name of the class.
2. Constructor should not return any value even void also (if we write the return type for the constructor then that constructor will be treated as ordinary method).
3. Constructors should not be static since constructors will be called each and every time whenever an object is creating.
4. Constructor should not be private provided an object of one class is created in another class (constructor can be private provided an object of one class created in the same class).
5. Constructors will not be inherited at all.
6. Constructors are called automatically whenever an object is creating.
7. A constructor cannot be abstract, static, final, native, strictfp, **or** synchronized.

Default Constructor

A constructor with no argument is known as **default constructor** in java. Default constructor provided by compiler if no-argument constructor is not written explicitly.

Ex.

```
class circle
{ private int r;
  public void setradius (int n)
    {r = n;
    }
  public void area ()
    {double a = 3.14 * r * r;
    System.out.println ("area is " + a);
    }
  public circle ()      //constructor
    {r = 1;
    }
}

class pc
{public static void main (String args [])
  {Circle a, b;
  a = new circle ();
  b = new circle ();
  a.setradius (7);
  b.setradius (2);
  a.area ();
  b.area ();
  }
}
```

Constructor with argument / Parameterized constructor

While creating an object we can pass some data. This data is passed on to the constructor as argument where it is used for initialization of the object.

Eg:-

```
class circle
{
    private int r;
    public void area ()
    {
        double a = 3.14 * r * r;
        System.out.println ("area is " + a);
    }
    public circle (int n)    // constructor with args.
    {
        r = n;
    }
}

class pc
{
    public static void main (String args [])
    {
        circle a, b;
        a = new circle (7);
        b = new circle (2);
        a.area ();
        b.area ();
    }
}
```

Constructor overloading

Defining multiple constructors in a class is called as constructor overloading. Only precaution is to be taken is that no. of args & type of args must be different.

Compiler decides which constructor to call depending on the no. of arguments & type of arguments which are pass while creating objects.

Eg:-

```
class circle
{ private int r;
  public void area ()
  {
    double a = 3.14 * r * r;
    System.out.println ("area is " + a);
  }
  public circle (int n)
  { r = n;
  }
  public circle ()
  { r = 1;
  }
}

class pc
{public static void main (String args [])
  {circle a, b, c;
   a = new circle (7);
   b = new circle (2);
   c = new circle ();
   a.area ();
   b.area ();
   c.area ();
  }
}
```

Constructor Vs Methods

| Java Constructor | Java Method |
|---|---|
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

private constructors

- Making the constructor private makes the class effectively final because a the sub-class can't access it's constructor.
- When a class needs to prevent the caller from creating objects. Private constructors are suitable. Objects can be constructed only internally.
- One application is in the singleton design pattern. The policy is that only one object of that class is supposed to exist. So no other class than itself can access the constructor. This ensures the single instance existence of the class.

Instance initializer block

Instance Initializer block is used to initialize the instance data member. It runs each time when an object of the class is created. The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Suppose we have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Instance initializer block is invoked at the time of object creation. The Java compiler copies the instance initializer block in the constructor after the first statement `super()`. So firstly, constructor is invoked.

Note: The Java compiler copies the code of instance initializer block in every constructor.

```
public class Test {
    int nonStaticVariable;
    // Instance initialization block: Runs before the constructor
    // each time you instantiate an object
    {
        System.out.println("Instance initialization.");
        nonStaticVariable = 7;
    }
    public Test() {
        System.out.println("Constructor.");
    }

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

Static Data Member

If a data member of a class is declared as static then only one copy of that data member is created for entire class & all the object of that class can share that data.

Eg :-

```
class circle
{ private int r; // object member or instance
  private static double pi = 3.14; // class member
  public void area ()
  { double a = pi * r * r;
    System.out.println ("area is " + a);
  }
  public circle (int n)
  {
    r = n;
  }
}
class pc
{
  public static void main (String args [])
  {
    circle a, b, c;
    a = new circle (7);
    b = new circle (2);
    c = new circle (5);
    a.area ();
    b.area ();
    c.area ();
  }
}
```


Static member functions

If a member function of a class is declared as static then such a function can be called without creating object of that class. Functions which does not have any relation with an Object must be declared as static .

Syntax:- **classname.memberfunction (args)**

NOTE :

- They can only call static methods within from them.
- Java static methods can only access static data members or fields.
- Java static methods cannot refer this and super

Eg:- class Account

```

    { private int accno, balance;
      private static double Irate = 10.75;
      public Account (int an, int b)
        { Accno = an;
          balance = b;
        }
      public void interest (int n)
        { double I = balance * irate * n /100.0;
          System.out.println ("Interest is " + i);
        }
      public static void changerate (double r)
        { Irate = r;
        }
    }
class pc
    { public static void main (String args [])
      { Account.changerate (11.75);
        Account a, b;
        a = new Account (4117, 5000);
        b = new Account (3527, 10000);
        a.intrest (3);
        b.intrest (5);
      }
    }

```

Static Initializers

If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes *static initialization blocks*.

A static initializer block resembles a method with no name, no arguments, and no return type. There is no need to refer to it from outside the class definition.

Syntax :

```
static
{
    //CODE
}
```

The code in a static initializer block is executed by the virtual machine when the class is loaded. Because it is executed automatically when the class is loaded, parameters don't make any sense, so a static initializer block doesn't have an argument list.

Static initializer is a code block with a name 'static'. A class can contain one or more static initializer blocks. The code in the static initializer block will be executed first followed by the code in the constructor, exactly once in the order they appear in the class. While the class is loaded, the executions in the static initializers take place.

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

Example :

```
class Test
{
    static int stNumber;
    int number;
    Test()
    {
        number = 10;
    }
    static
    {
        stNumber=30;
    }
    ..... // other methods here
}
```

In the above example, the code in static initializer will be executed first followed by the code in the constructor.

Instance Methods Vs Class Methods

| Instance method | Class method |
|---|--|
| Instance method can be called without creating object | Class method can be called with the object |
| Instance method can called only other static method | Class method can called other method and static method |
| Instance method is also called as static method | Class method is also called as member function |
| Instance method is called with class name | Class method is called with object name |

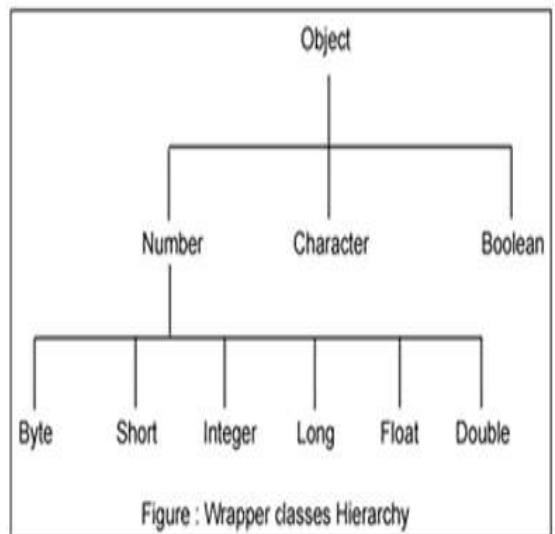
Wrapper class /Numbers Classes

Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

All of the numeric wrapper classes are subclasses of the abstract class `Number`:

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



class Integer

It is an wrapper class which wraps a value of type int in it.

Constructor:-

1) Integer (int n):-

Will create an integer object from int value.

Eg:- Integer b;
int a =5;
b = new Integer (a);

2) Integer (String n):-

Will create an integer object from the String value.

Eg:- Integer b;
b = new Integer ("5");

Method:-

1) int intValue ():-

Return the int value from object.

Eg :- int c = b.intValue () + 7;

2) long longValue ():-

3) float floatValue ():-

4) double doubleValue ():-

5) public static int parseInt (String n):-

It converts String value into an int value

6) public static String toBinaryString (int n):-

Returns binary of given int value.

Eg:- String s = Integer.toBinaryString (5);

7) public static String toHexString (int n):-

It hexa decimal value.

8) public static String toOctalString (int n):-

It returns octal of given value.

*** WAP to read a int no & count total no of 1's in to binary?**

```
import javax.swing.*;
class Binaryno
{
    public static void main(String arg[])
    {
        int C=0,d=0
        String s=JOptionPane.showInputDialog(null, "Enter no");
        int num=Integer.parseInt(s);
        String s2=Integer.toString (num);
        System.out.println ("Binary value "+s1);
        for(int I =1;I <b.length();I ++)
            {
                char ch = b.charAt(I);
                if(ch=='1')
                    C++;
            }
        System.out.println("total no =" +d);
    }
}
```

*** WAP to print all nos from 1 to 25 along with its Hex & Oct.**

```
class sum
{
    public static void main(String arg[])
    {
        for(int I=1;I<=25;I++)
        {
            String s1=Integer.toHexString(I);
            String s2=Integer.toOctalString(I);
            System.out.println(I+" "+s1+" "+s2);
        }
    }
}
```

Class Character

It is wrapper class which wraps value of char in it. It provides us different static method to perform operations on char type of data.

Method :-

- 1) **public static boolean isDigit (char ch):-**
Returns true if a given character represents a digits.
Eg:- boolean b= Character.isDigit (7);
- 2) **public static boolean isLetter (char ch):-**
Returns true if a given character represents alphabet or not.
Eg:- boolean b= Character.isLetter ('7');
- 3) **public static boolean isSpace (char ch):-**
Returns true if a given character represents a ' ', '\t', '\n', '\r'
- 4) **public static boolean isUpperCase (char ch):-**
Returns true if a given character is in upper case.
- 5) **public static boolean isLowerCase (char ch):-**
Returns true if a given character is in lower case.
- 6) **public static char toUpperCase (char ch):-**
Converts given character into upper case.
- 7) **public static char toLowerCase (char ch):-**
Converts given character into lower case.

* WAP to count total no. of digits in given String.

```
class pc
{
    public static void main (String arg[])
    {
        int c = 0;
        String s = "MH27H6390"
        for (int I =1; I < s.length(); I ++)
        {
            char ch=s.charAt (I);
            boolean b = Character.isDigit (ch);
            if(b == True )
                c++;
        }
        System.out.println ("Total Digit is "+ c);
    }
}
```

The finalize() Method:

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize()**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize() to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize() method. The Java runtime 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.

The finalize() method has this general form:

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

Here, the keyword protected is a specifier that prevents access to finalize() by code defined outside its class.

This means that you cannot know when or even if finalize() will be executed. For example, if your program ends before garbage collection occurs, finalize() will not execute.

Arrays

An array is a group of elements of same type sharing same name. In java arrays are treated like an object. They can be dynamically created by using a keyword **new**.

Array Reference:

It is a variable in which we can store ID of an Array.

Syntax : *datatype [] arrayname;* or
 datatype arrayname[];

Creating Array:

An array can be dynamically created by using keyword **new**, specifying datatype & no. of elements.

Syntax : *arrayname =new datatype[size];*

NOTE: whenever an array is created space is reserved for its elements & a special member length which contains size of array.

Array Size:

Size of array can be find out by using its special public member **length**

yntax : *arrayname.length*

```

Eg.  class pc
      {
      public static void main (String arg [])
      {
          {int a[ ];           // declare a array reference
          a=new int[5];        // create an array
          a[0]=244;            // initialize it.
          a[1]=4;
          a[2]=76;
          a[3]=6544;
          a[4]=42;

                                  // print the array
          for(int i=0;i< a.length;i++)
              System.out.println(a[i]);
      }
      }

```

Array Initialization

An array can be initialized while creating it.

Syntax : *datatype arrayname[] = { list of values.....};*

Eg. `int a[]={544, 322, 86, 332, 667, 21, 366, 22};`

*** WAP to find sum of all elements of an array.**

```

class pc
{
    public static void main (String arg [])
    {
        int a[ ]={544, 322, 86, 332, 667, 21, 366, 22};

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

        System.out.println("Sum is "+s);
    }
}

```

*** WAP to find Average temp if temp values are given for a week in an array.**

class pc

```
{public static void main (String arg [])
{
    double a[ ]={35.44, 32.2, 28.6, 33.2, 26.67, 21.9, 36.6};

    double s=0;
    for(int i=0;i< a.length;i++)
        s=s+a[i];
    double m=s/a.length;
    System.out.println("Average temp is "+m);
}
}
```

*** WAP to find greatest of all elements from an array.**

```
class pc
{
    public static void main (String arg [])
    {
        int a[ ]={544, 322, 86, 332, 667, 21, 366, 22};

        int g=a[0];
        for(int i=1;i< a.length;i++)
            if(a[i]>g)
                g=a[i];

        System.out.println("Greatest no is "+g);
    }
}
```

Multidimensional Array

In java a multidimensional array is actually an array of an array. It creates multiple single dimensional array. These array may be of same size or of different size.

Eg.

```
int a[ ][ ];
a=new int[2][4];    // it will create two single dimensional array of size 4
```

this array will have 2 rows of size 4 each.

| | | | |
|--|--|--|--|
| | | | |
| | | | |

Eg.

```
int a[ ][ ];
a=new int [2][ ];
a[0]=new int [5];    // it will create two single dimensional array of size5.
a[1]=new int [3];    // it will create two single dimensional array of size3.
```

this array will have 2 rows. First row of size 5 & second row of size 3.

| | | | | |
|--|--|--|--|--|
| | | | | |
| | | | | |

Array Initialization

An array can be initialized while creating it.

Syntax : *datatype arrayname[][] = {{ list of values.....},{ list of values.....},...};*

Eg. `int a[][] = { { 54, 4, 32, 2, 8, 6, 3, 32 },
 { 6, 67, 21, 366 },
 { 2, 6, 5, 5, 2, 3 } };`

*** WAP to find sum of all elements of an array.**

```
class pc
{
    public static void main (String arg [ ])
    {
        int a[ ][ ] = { { 54, 4, 32, 2, 8, 6, 3, 32 },
                        { 6, 67, 21, 366 },
                        { 2, 6, 5, 5, 2, 3 } };
        int s=0;
        for(int i=0;i< a.length;i++)
            for(int j=0;j< a[i].length;j++)
                s=s+a[i][j];
        System.out.println("Sum is "+s);
    }
}
```

*** WAP to find sum of each row of an array.**

```
class pc
{
    public static void main (String arg [ ])
    {
        int a[ ][ ] = { { 54, 4, 32, 2, 8, 6, 3, 32 },
                        { 6, 67, 21, 366 },
                        { 2, 6, 5, 5, 2, 3 } };
        for(int i=0;i< a.length;i++)
        {
            int s=0;
            for(int j=0;j< a[i].length;j++)
                s=s+a[i][j];
            System.out.println("Sum of row " + i + " is "+s);
        }
    }
}
```

Array of Objects

Whenever an array of object is created space is created by references of Objects & not Objects

| | |
|-------------------------------|---------------------------------------|
| Syntax : To declare Reference | <i>className arrayName[];</i> |
| To Create Array | <i>arrayName=new className[size];</i> |

```

class circle
{private int r;
  public void area ()
  {double a = 3.14 * r * r;
   System.out.println ("area is " + a);
  }
  public circle (int n)
  {r = n;
  }
}

class pc
{ public static void main (String args [])
  { circle a [ ];           // defining a array reference
    a = new circle [5];     // creating 5 references for array object
    a[0] = new circle (2);  // creating 5 objects
    a[1] = new circle (7);
    a[2] = new circle (5);
    a[3] = new circle (12);
    a[4] = new circle (8);
    for(int i=0;i< a.length;i++) // calling fn area for all objects
      a[i].area ();
  }
}

```

Class Arrays

The `java.util.Arrays` class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

| SN | Methods with Description |
|----|--|
| 1 | <code>public static int binarySearch(Object[] a, Object key)</code> Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, <code>-(insertion point + 1)</code> . |
| 3 | <code>public static void fill(int[] a, int val)</code> Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other primitive data types (Byte, short, Int etc.) |
| 4 | <code>public static void sort(Object[] a)</code> Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other primitive data types (Byte, short, Int, etc.) |

```
import java.util.*;

class demo
{
    public static void main(String args[ ])
    {
        int arr[]={23,5,12,111,15,6,2};
        Arrays.sort(arr);
        display(arr);

        int pos=Arrays.binarySearch(arr,7);
        if(pos<0)
            System.out.println("Not Found");
        else
            System.out.println("Found at pos :"+pos);
    }
    public static void display(int arr[])
    {
        for(int n:arr)
            System.out.println(n);
    }
}
```


Variable Arguments(var-args):

Java enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

```
public class VarargsDemo {
    public static void main(String args[]) {
        // Call method with variable args
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }
    public static void printMax( double... numbers) {
        if (numbers.length == 0) {
            System.out.println("No argument passed");
            return;
        }

        double result = numbers[0];

        for (int i = 1; i < numbers.length; i++)
            if (numbers[i] > result)
                result = numbers[i];
        System.out.println("The max value is " + result);
    }
}
```

Pass by Value

When the argument is of primitive type , pass-by-value means that the method cannot change its value. When the argument is of reference type, pass-by-value means that the method cannot change the object reference, but can invoke the object's methods and modify the accessible variables within the object.

```
class CCIT
{
    public static void change(int a)
    {
        a++;
    }
}

class pc
{
    public static void main(String arg[])
    {
        int a=5;
        CCIT.change(a);
        System.out.println(a);
    }
    // Result of this program will be 5 and not 6
    // as while calling
    //method change value of a is passed as argument.
```

Pass by Reference

For a method to modify an argument, it must be of a reference type such as an object or array. Objects and arrays are passed by reference. So the effect is that arguments of reference types are passed in by reference. Hence the name. A reference to an object is the address of the object in memory. Now, the argument in the method is referring to the same memory location as the caller.

| | |
|---|--|
| <pre> class Circle { private int R; public void setRadius(int n) { R=n; } public int getRadius() { return R; } } </pre> | <pre> class CCIT { public static void Change(Circle b) {b.setRadius(2); } } </pre> |
|---|--|

class pc

```

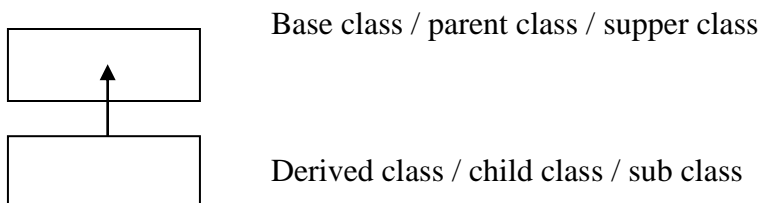
{public static void main(String arg[])
    { Circle a;
      a=new Circle();
      a.setRadius(5);
      CCIT.change(a);
      System.out.println(a.getRadius());
    } // Result of this program will be 2 and not 5 as while calling
      //method change reference of object 'a' is passed as argument.
} // so the changes made by the method are actually done on the //original object

```

Inheritance

Inheritance is the ability, offered by many OOP languages, to derive a new class (the *derived* or *inherited* class) from another class (the *base* class). The derived class automatically inherits the properties and methods of the base class. For example, you could define a generic *Shape* class with properties such as *Color* and *Position* and then use it as a base for more specific classes (for example, *Rectangle*, *Circle*, and so on) that inherit all those generic properties. You could then add specific members, such as *Width* and *Height* for the *Rectangle* class and *Radius* for the *Circle* class. It's interesting to note that, while polymorphism tends to reduce the amount of code necessary to use the class, inheritance reduces the code inside the class itself and therefore simplifies the job of the class creator.

Inheritance is also sometimes called **generalization**, because the **is-a** relationships represent a hierarchy between classes of objects. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple **is-a** fruit), apples may naturally inherit all the properties common to all fruit, such as being a fleshy container for the seed of a plant.



Syntax:- `class classname extends baseclassname`
 {
 Data member
 Member function.
 }

By using this technique we can reuse our existing code.

```
For eg:  class Account
        { protected int accno, balance;
        public void open (int an, int b)
            { Accno = an;
            balance = b;
            }
        public void deposit (int amt)
            { balance = balance + amt;
            }
        public void withdraw (int amt)
            { balance = balance - amt;
            }
        public void showBalance ()
            { System.out.println("Balance is =" + balance);
            }
        }
class SAccount extends Account
    {
    public void addIntrest (double r, int n)
        { double I = balance * r * n / 100;
        balance = balance + (int) I ;
        }
    }
```

class pc

```
{ public static void main (String arg [])
    { Account a = new Account();
    a.open (4117, 5000);
    a.withdraw (2000);
    a.showBalance ();
    SAccount b = new Saccount ();
    b.open (3210, 8000);
    b.deposit (7000);
    b.addIntrest (10.25, 3);
    b.showBalance ();
    }
}
```

Method overriding

Redefining a base class method in derived class is called as method overriding.

The overriding method must have same name, same no. & type of arguments similar to a base class method. So when the method is called for the derived class objects this overridden method is executed instead of inherited base class method.

Rules for Method Overriding:

1. applies only to inherited methods
2. object type (NOT reference variable type) determines which overridden method will be used at runtime
3. Overriding method can have different return type.
4. Overriding method must not have more restrictive access modifier
5. Abstract methods must be overridden
6. Static and final methods cannot be overridden
7. Constructors cannot be overridden
8. It is also known as Runtime polymorphism.

Design a class box containing data member's length, breadth & height & member function setdimension, value & surface_area then derived a new class openbox from class box.

```

class box
{ protected int l, b, h;
  public void setdimension (int x, int y, int z)
    { l = x;
      b = y;
      h = z;
    }
  public void volume ()
    { int v = l * b * h;
      System.out.println("Volume is "+ v);
    }
  public void surface_area ()
    { int a = 2 * l * h + 2 * h * b + 2 * b * l;
      System.out.println ("Surface area is "+ a);
    }
}

class openbox extends box
{ public void surface_area ()
    { int a = 2 * l * h + 2 * h * b + b * l;
      System.out.println ("Surface area is"+ a);
    }
}

class pc
{ public static void main (String arg [])
    { openbox a = new openbox ();
      a.setdimension (4, 7, 5);
      a.volume ();
      a.surfacearea ();
    }
}

```


Design a class a/c containing data member's accno & balance and member function open, deposit, withdraw & showbalance. Then derived a new class current a/c from a class a/c having the transaction.

```

class Account
{ protected int accno, balance;
  public void open (int an, int b)
    { accno = an;
      balance = b;
    }
  public void deposit (int amt)
    {balance = balance + amt;
    }
  public void withdraw (int amt)
    {balance = balance - amt;
    }
  public void showBalance ()
    {System.out.println("Balance is = "+ balance);
    }
}
class CAccount extends Account
{public void deposit (int amt)
  {balance = (balance + amt)- 1;
  }
  public void withdraw (int amt)
  {
    balance = (balance - amt)- 1;
  }
}

```

```

class pc
{
    public static void main (String arg [])
    {
        Account a = new Account();
        a.open (4117, 5000);
        a.withdraw (2000);
        a.deposit (7000);
        a.showBalance ();

        SAccount b = new Saccount ();
        b.open (3210, 8000);
        b.deposit (7000);
        b.withdraw (2000);
        b.showBalance ();
    }
}

```

Calling base overridden function in derived class

This can be done by using a keyword **super**.

super keyword

This keyword is used to access base class member in derived class if derived class acts as a reference of base class is derived class.

Eg:-

```

class CAccount extends Account
{
    public void deposit (int amt)
    {
super.deposit (amt);
        balance = (balance + amt)- 1;
    }

    public void withdraw (int amt)
    {
super.withdraw (amt);
        balance = (balance - amt)- 1;
    }
}

```

Overloading vs Overriding in Java

1. Overloading happens at **compile-time** while Overriding happens at **runtime**: The binding of overloaded method call to its definition happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. **Static binding** is being used for overloaded methods and **dynamic binding** is being used for overridden / overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type .
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

super keyword in java

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

this keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

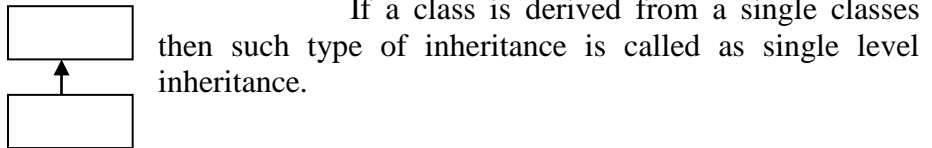
Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

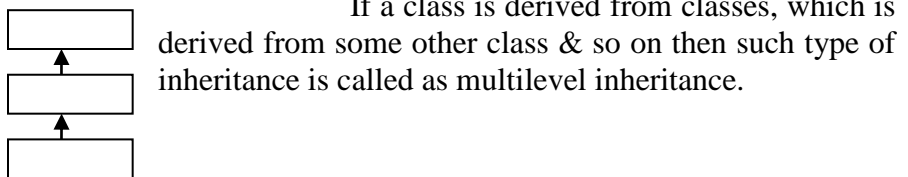
Type of Inheritance

We can derived our classes in different way

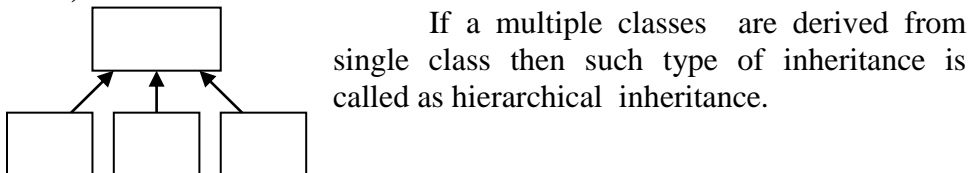
1) Single level inheritance :-



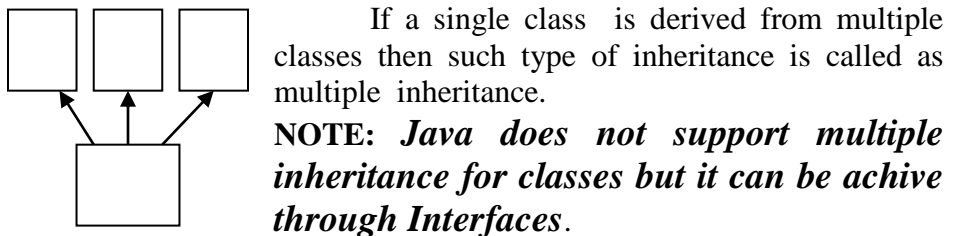
2) Multi level inheritance:-



3) Hierarchical inheritance:-



4) Multiple inheritance:-



Design class person containing data member name address and member function containing setdata and showdata then derived a new class employee from person containing additional data member job and salary

class person

```
{protected String Name, Add;
public void setdata(String n, String a)
    {Name=a;
    Add=a;
    }
public void showdata()
    {System.out.println ("Name="+Name);
    System.out.println ("Add="+Add);
    }
}
```

class employee extends person

```
{protected String Job;
protected int Sal;
public void setdata(String n, String a, String j, int s)
    { super.setdata (n, a);
    Job =j;
    Sal =s;
    }
public void showdata ()
    {super.showdata ();
    System.out.println ("Job is="+Job);
    System.out.println ("Salary is="+Sal);
    }
}
```

class pc

```
{ public static void main(String arg[])
    {
    employee b;
    b=new employee("Amit Jain",23,"Manager",34500);
    b.showdata();
    }
}
```

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods* ie., methods with out body (public void get();)
- But, if a class have at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract it cannot be instantiated.
- To use an abstract class you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class you have to provide implementations to all the abstract methods in it.

Abstract Classes and Methods

If a method is just declared & not defined then such a method is called as abstract method. A class which contains a abstract method is called as abstract class.

Eg.

```
abstract class Circle
{
    protected int R;
    public void setRadius(int n)
    {
        R=n;
    }
    public void area()
    {
        double a=3.14.*R*R;
        System.out.println("Area is "+a);
    }
    abstract public void circumference( ); // this is an abstract method
}
```

// it is declared but not defined.

We cannot create object of an abstract class. Such classes are defined to achieve polymorphism through dynamic linking. Multiple classes can be derived from an abstract class. If a class is derived from an abstract class then it must override its abstract methods otherwise the derived class will also become an abstract class.

Aggregation

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

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

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

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

When to use Aggregation

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

```
public class Address {
String city,state,country;
```

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

```
}
}
```

```
public class Emp {
int id;
String name;
Address address;
```

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

```
void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.countr
y);
}
```

```
public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
```

```
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
```

```
e.display();
e2.display();
}
}
```

Final Keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

2) Java final method

If you make any method as final, you cannot override it.

3) Java final class

If you make any class as final, you cannot extend it.

Strictfp Keyword

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

```
strictfp class A{}//strictfp applied on class
```

```
strictfp interface M{}//strictfp applied on interface
```

```
class A{  
strictfp void m(){}//strictfp applied on method  
}
```

Polymorphism

Polymorphism is the ability of different classes to expose similar (or identical) interfaces to the outside. This similarity simplifies your job as a programmer because you don't have to remember hundreds of different names and syntax formats. We create generic procedures that act on all the different Objects and therefore noticeably reduce the amount of code you have to write.

Polymorphism is useful in that it allows Java programs to be written more abstractly, and more abstraction allows for more efficiency and less redundancy. Imagine you are working with a family of classes, and a certain program requires use of one of the subclasses, but it will not know which subclass to use until runtime. In this case, you can use a superclass variable throughout the program, and point it to the corresponding subclass when the time is right. Polymorphism is also useful when using certain methods throughout different families of classes. In general, polymorphism allows inheritance and interfaces to be used more abstractly.

To achieve polymorphism through dynamic linking Java provides a feature i.e. **" A base class reference can point to object of its derived class " but through that reference we can access members which are declared in base class only .**

```
abstract class Shape
{
    abstract public void area( );
}

class Circle
{
    private int R;

    public Circle(int n)
    {
        R=n;
    }

    public void area( )
    {
        double a=3.14*R*R;
        System.out.println("Area is "+a);
    }
}
```

```

class Rect
{ private int L,B;
public Rect(int m,int n)
    {L=m;
    B=n;
    }
public void area( )
    { int a=L*B;
    System.out.println("Area is "+a);
    }
}

```

```

class Test
{public static void main(String args[ ])
    {Shape p[ ]=new Shape[5]; //create 5 references of type shape.
    p[0]=new Circle(4);      // a shape reference can point to a Circle object
    p[1]=new Rect(3, 8);     // as well as a Rect object
    p[2]=new Rect(7, 4);
    p[3]=new Circle(11);
    p[4]=new Rect(5, 5);
    for(int i=0;i< p.length;i++)
        p[i].area(); // method area of Circle or Rect will be called
    }                          // depending on type of the object.
}

```

instanceof Operator

This operator is used to datatype of a Object pointed by a reference.

Syntax : *reference instanceof typename*

This exp returns true if the object is of specified type

```

if ( p instanceof Circle)
    .....
if(p instanceof Rect)
    .....

```

Compile time Vs Run time Polymorphism

| <i>Compile time Polymorphism</i> | <i>Run time Polymorphism</i> |
|---|--|
| In Compile time Polymorphism, call is resolved by the compiler . | In Run time Polymorphism, call is not resolved by the compiler. |
| It is also known as Static binding, Early binding and overloading as well. | It is also known as Dynamic binding, Late binding and overriding as well. |
| Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type. | Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass. |
| It is achieved by function overloading and operator overloading. | It is achieved by default when a base class reference points to derived class Object. |
| It provides fast execution because known early at compile time. | It provides slow execution as compare to early binding because it is known at runtime. |
| Compile time polymorphism is less flexible as all things execute at compile time. | Run time polymorphism is more flexible as all things execute at run time. |

Interface

Definition: An *interface* is a named collection of method definitions (without implementations). An interface can also declare constants.

An *interface* defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior. Because an interface is simply a list of unimplemented, and therefore abstract, methods, you might wonder how an interface differs from an abstract class. The differences are significant.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Class Vs Interfaces

| Property | Class | Interface |
|---------------|---|---|
| Instantiation | Can Be Instantiated | Can not be instantiated |
| State | Each Object created will have its own state | Each objected created after implementing will have the same state |
| Behavior | Every Object will have the same behavior unless overridden. | Every Object will have to define its own behavior by implementing the contract defined. |
| Inheritance | A Class can inherit only one Class and can implement many interfaces | An Interface cannot inherit any classes while it can extend many interfaces |
| Variables | All the variables are instance by default unless otherwise specified | All the variables are static final by default, and a value needs to be assigned at the time of definition |
| Methods | All the methods should be having a definition unless decorated with an abstract keyword | All the methods are abstract by default and they will not have a definition. |

Abstract class Vs Interfaces

| Abstract class | Interface |
|---|--|
| 1) Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. |
| 2) Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| 4) Abstract class can have static methods, main method and constructor. | Interface can't have static methods, main method or constructor. |
| 5) Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| 6) The abstract keyword is used to declare abstract class. | The interface keyword is used to declare interface. |
| 7) Example: <pre>public abstract class Shape { public abstract void draw(); }</pre> | Example: <pre>public interface Drawable { void draw(); }</pre> |

Defining an Interface

An interface definition has two components: the interface declaration and the interface body. The interface declaration declares various attributes about the interface, such as its name and whether it extends other interfaces. The interface body contains the constant and the method declarations for that interface.

Syntax :

```
interface Name
    { // final Data Members
      // abstract Member Functions
    }
```

For eg:

```
interface Solid
{
    final static double PI=3.1417;
    public void volume();
    public void surfaceArea();
}
```

This interface defines one constants PI. This interface also declares, but does not implement, the volume() & surfaceArea() method. Classes that implement this interface provide the implementation for that method.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final.

Implementing an Interface

An interface defines a protocol of behavior. A class that implements an interface adheres to the protocol defined by that interface. To declare a class that implements an interface, include an `implements` clause in the class declaration. Your class can implement more than one interface (the Java platform supports multiple inheritance for interfaces), so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class.

By Convention: The `implements` clause follows the `extends` clause, if it exists.

```
class Sphere implements Solid
{
    .....    //in this class we have to override volume() and surfaceArea () method
              //otherwise this class will become an abstract class
}
```

When a class implements an interface, it is essentially signing a contract. Either the class must implement all the methods declared in the interface and its superinterfaces, or the class must be declared `abstract`. The method signature--the name and the number and type of arguments in the class--must match the method signature as it appears in the interface

Interface Features:

- Variable of an interface are explicitly declared final and static (as constant) meaning that the implementing the class cannot change them they must be initialize with a constant value all the variable are implicitly public of the interface, itself, is declared as a public .
- Method declaration contains only a list of methods without anybody statement and ends with a semicolon the method are, essentially, abstract methods there can be default implementation of any method specified within an interface each class that include an interface must implement all of the method.

Need:

- To achieve multiple Inheritance.
- We can implement more than one Interface in the one class.
- Methods can be implemented by one or more class.
- To Define a contract which must be implemented by class.
- To Achive Multiple Inheritance

Default Methods In Java

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

// A simple program to Test Interface default methods in java
interface TestInterface

```
{    // abstract method
    public void square(int a);

    // default method
    default void show()
    { System.out.println("Default Method Executed");
    }
}
```

class TestClass implements TestInterface

```
{
    // implementation of square abstract method
    public void square(int a)
    {    System.out.println(a*a);
    }

    public static void main(String args[])
    { TestClass d = new TestClass();
      d.square(4);

      // default method executed
      d.show();
    }
}
```

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class. Given below is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

```
class Outer_Demo{
    int num;
    //inner class
    private class Inner_Demo{
        public void print(){
            System.out.println("This is an inner class");
        }
    }
    //Accessing the inner class from the method within
    void display_Inner(){
        Inner_Demo inner = new Inner_Demo();
        inner.print();
    }
}

public class My_class{
    public static void main(String args[]){
        //Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();
        //Accessing the display_Inner() method.
        outer.display_Inner();
    }
}
```


Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows:

```
AnonymousInner an_inner = new AnonymousInner(){  
    public void my_method(){  
        .....  
        .....  
    }  
};
```

The following program shows how to override the method of a class using anonymous inner class.

```
abstract class AnonymousInner{  
    public abstract void mymethod();  
}  
  
public class Outer_class {  
    public static void main(String args[]){  
        AnonymousInner inner = new AnonymousInner(){
```

```
        public void mymethod(){  
            System.out.println("This is an example of anonymous  
inner class");  
        }  
    };  
    inner.mymethod();  
}  
}
```

If you compile and execute the above program, you will get the following result.

```
This is an example of anonymous inner class
```

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument:

```
obj.my_Method(new My_Class(){  
    public void Do(){  
        .....  
        .....  
    }  
});
```

The following program shows how to pass an anonymous inner class as a method argument.

```
//interface  
interface Message{  
    String greet();  
}
```

```
public class My_class {  
    //method which accepts the object of interface Message  
    public void displayMessage(Message m){  
        System.out.println(m.greet() +", This is an example of  
anonymous inner class as an argument");  
    }  
  
    public static void main(String args[]){  
        //Instantiating the class  
        My_class obj = new My_class();  
  
        //Passing an anonymous inner class as an argument  
        obj.displayMessage(new Message(){  
            public String greet(){  
                return "Hello";  
            }  
        });  
    }  
}
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows:

```
class MyOuter {  
    static class Nested_Demo{  
    }  
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

```
public class Outer{  
    static class Nested_Demo{  
        public void my_method(){  
            System.out.println("This is my nested class");  
        }  
    }  
    public static void main(String args[]){  
        Outer.Nested_Demo nested = new Outer.Nested_Demo();  
        nested.my_method();  
    }  
}
```

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method. A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

```
public class Outerclass{
    //instance method of the outer class
    void my_Method(){
        int num = 23;
        //method-local inner class
        class MethodInner_Demo{
            public void print(){
                System.out.println("This is method inner class
"+num);
            }
        }//end of inner class
        //Accessing the inner class
        MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }
    public static void main(String args[]){
        Outerclass outer = new Outerclass();
        outer.my_Method();
    }
}
```

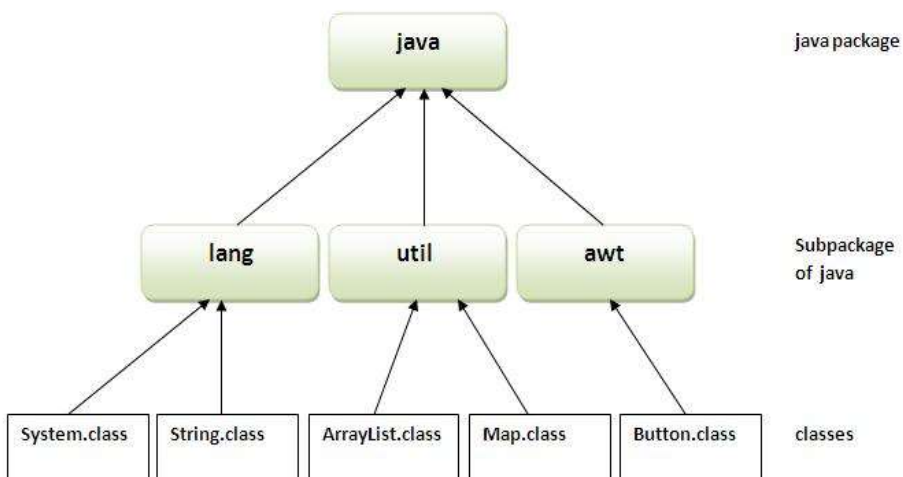
Packages

To make classes easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes and interfaces into packages.

Definition: A *package* is a collection of related classes and interfaces providing access protection and namespace management.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.

You should bundle these classes and the interface in a package for several reasons:

- You and other programmers can easily determine that these classes and interfaces are related.
- You and other programmers know where to find classes and interfaces that provide graphics-related functions.
- The names of your classes won't conflict with class names in other packages, because the package creates a new namespace.
- You can allow classes within the package to have unrestricted access to one another yet still restrict access for classes outside the package.

access package from another package

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. fully qualified name.

The import Keyword

import keyword is used to import built-in and user-defined packages into your java source file. So that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to class that is present in different package

1. **Using fully qualified name** (But this is not a good practice.)

Example :

```
class MyDate extends java.util.Date
{
    //statement;
}
```

2. **import the only class you want to use.**

Example :

```
import java.util.Date;
class MyDate extends Date
{
    //statement.
}
```

3. **import all the classes from the particular package**

Example :

```
import java.util.*;
class MyDate extends Date
{
    //statement;
}
```

static import

static import is a feature that expands the capabilities of **import** keyword. It is used to import **static** member of a class. We all know that static member are referred in association with its class name outside the class. Using **static import**, it is possible to refer to the static member directly without its class name. There are two general form of static import statement.

- The first form of **static import** statement, import only a single static member of a class

Syntax

```
import static package.class-name.static-member-name;
```

Example

```
import static java.lang.Math.sqrt;    //importing static method
sqrt of Math class
```

- The second form of **static import** statement, imports all the static member of a class

Syntax

```
import static package.class-type-name.*;
```

Example

```
import static java.lang.Math.*;        //importing all static
member of Math class
```

Java Library Packages

java.lang

Java.lang package contains the classes that are fundamental to the design of the Java programming language. This package is imported by default.

For eg: Maths, String, System, Integer, Class, Object etc

java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

For eg: Date, Random, Vector, Set , Map .etc.

java.awt

Contains all of the classes for creating user interfaces and for painting graphics and images.

For eg: Font, Color, Image, Button, TextField..etc.

java.io

Java.io package provides classes for system input and output through data streams, serialization and the file system.

For eg: File , InputStream,OutputStream,Reader,Writer etc.

java.net

contains classess for Communication and Netwroking.

For eg: InetAddress, URL , Socket, DatagramPacket etc.

class String

An object of this type is used to store a strings. Data stored in a string object can not be modified.

Constructors :-

1) **String (char a []):-**

Creates a string object from a character array.

Eg:- `char a[] = {'C', 'C', 'I', 'T'};`
`String s= new String(a);`
`System.out.println (s);` O/p :- “CCIT”

2) **String (char a [], int offset, int n):-**

Will create a string object from character array by using **n** character starting from giver offset.

Eg:- `char a[] = {'C', 'C', 'I', 'T'};`
`String s= new String(a, 1, 2);`
`System.out.println (s);` O/p: - “CI”

3) **String (byte b []):-**

It will create a string object from given byte array.

Eg:- `byte b [] = {65, 66, 67, 68, 69, 70, 71, 72};`
`String s= new String (b);`
`System.out.println (s);` O/p: - “ABCDEFGH”

4) **String (byte b [], int offset, int n):-**

It will create a string object by using an object starting by given offset..

Eg:- `byte b [] = {65, 66, 67, 68, 69, 70, 71, 72};`
`String s= new String (b, 2, 2);`
`System.out.println (s);` O/p: - “CD”

5) **String (String str):-**

Will create a string object while using data of another string.

Eg:- `String s= new String (“CD”);`
`System.out.println (s);` O/p: - “CD”

NOTE: String object are automatically created when ever java founds strings in double codes.

```
Eg:- String s;
      s = "Amravati";
      System.out.println (s);      O/p: - "Amravati"
```

Methods:-

1) int length ():-

It returns length of the string.

```
Eg:- String s;
      s = "Amravati";
      int n = s.length();
      System.out.println ("Length is " + n);
```

2) char charAt (int index):-

It returns character from given position.

```
Eg:- String s;
      s = "Amravati";
      char ch = s.charAt(2);
      System.out.println (ch);
```

3) String toUpperCase ():-

It returns a new string object converted to upper case.

```
Eg:- String s, p;
      s = "amravati";
      p = s.toUpperCase ();
      System.out.println (p);
```

4) String toLowerCase ():-

It returns a new string object converted to lower case.

Eg:-

```
String s, p;  
s = "AMRAVATI";  
p = s.toLowerCase ();  
System.out.println (p);
```

5) String subString (int beginindex):-

It returns sub string starting from given position.

Eg:-

```
String s, p;  
s = "Amravati";  
p = s.subString (3);  
System.out.println (p);
```

6) String subString (int beginindex, int endindex):-

It returns sub string starting from begin index up to end index.
Including beginindex and excluding endindex.

Eg:-

```
String s, p;  
s = "Amravati";  
p = s.subString (2, 4);  
System.out.println (p);
```

7) boolean endsWith (String pattern):-

It returns true if the string ends with given pattern.

Eg:-

```
String s;  
s = "Amravati";  
boolean b = s.endsWith ("at");  
System.out.println (b);
```

8) boolean startsWith (String pattern):-

It returns true if the string starts with given pattern.

Eg:-

```
String s;
s = "Amravati";
boolean b = s.startsWith ("mr");
System.out.println (b);
```

9) boolean startsWith (String pattern, int offset):-

It returns true if the pattern is present at given offset.

Eg:-

```
String s;
s = "Amravati";
boolean b = s.startsWith ("mr", 1);
System.out.println (b);
```

10) String replace (char oldchar, char newchar):-

It replaces all the occurrences of old char with new char.

Eg:-

```
String s, b;
s = "Amravati";
b = s.replace ('a', 'z');
System.out.println (b);
```

11) boolean equals (String obj):-

It returns true if data of both the string object is same.

Eg:-

```
String s1, s2;
s1 = new String ("ram");
s2 = new String ("ram");
if (s1.equals(s2))
    System.out.println ("Same");
Else
    System.out.println ("Different");
```

12) boolean equalsIgnoreCase (String obj):-

It compare two string without case sensitivity.

Eg:- String s1, s2;
 s1 = new String ("ram");
 s2 = new String ("ram");
 if (s1.equalsIgnoreCase (s2))
 System.out.println ("Same");
 Else
 System.out.println ("Different");

13) int compareTo (String obj):-

This compare two string & return compare between their ASCII values. (return 0 = both same)

Eg:- String s1, s2;
 s1 = new String ("ram");
 s2 = new String ("ram");
 if (s1.compareTo(s2)==0)
 System.out.println ("Same");
 else
 System.out.println ("Different");

14) String trim ():-

It removes leading & trailing blank spaces from the string.

Eg:- String s1, s2;
 s1 = new String ("ram shyam");
 s2 = s1.trim ();
 System.out.println (s2);

15) int indexOf (char ch):-

It search character in the string & if it found it return it returns it position & if not found it return '-1'

Eg:- String s1= new String ("Amravati");
 int n = s1.indexOf ('a');
 if (n == -1)
 System.out.println ("Not found");
 else
 System.out.println ("Found at position "+n);

16) int indexOf (char ch, int offset):-

It will start searching from given offset

Eg:-

```
String s1;
s1 = new String ("Amravati");
int n = s1.indexOf ('m', 2);
if (n == -1)
    System.out.println ("Not found");
else
    System.out.println ("Found at position "+n);
```

17) int indexOf (String pattern):-

Will search the pattern in the string.

Eg:-

```
String s1;
s1 = new String ("Amravati");
int n = s1.indexOf ("rm");
System.out.println (n);
```

18) int indexOf (String pattern, int offset):-

Will start searching from given offset.

19) int lastIndexOf (char ch):-

It will start searching from end in backward direction

Eg:-

```
String s1 = new String ("Amravati");
int n = s1.lastIndexOf ('a');
System.out.println (n);
```

20) int lastIndexOf (char ch, int offset):-

It will start searching from given offset in backward direction

Eg:-

```
String s1 = new String ("Amravati");
int n = s1.lastIndexOf ('a', 2);
System.out.println (n);
```

21) int lastIndexOf (String pattern):-

It will search the pattern last position in backward direction.

Eg:- String s1 = new String ("Amravati");
 int n = s1.lastIndexOf ("ati");
 System.out.println (n);

22) int lastIndexOf (string pattern, int offset):-

Will search the pattern from last position in backward direction from given offset.

Static Methods of class String**23) static String valueOf (int value):-**

Convert integer value into string.

24) static String valueOf (long value):-

Convert long value into string.

Similar methods for different datatypes.....

WAP to count total no. of a's in given string

class count

```
{ public static void main (String arg [])
    { int c=0;
      String s = "Madam Mohan";
      for (int I = 0; I < s.length; I ++)
      {
        char ch = s.charAt (i);
        if (ch = 'a' || ch = 'A')
          c++;
      }
      System.out.println("Total a's is" + c);
    }
}
```

WAP to read a sentence & check if the ‘CCIT’ is present in that sentence or not.

```
class pc
{
    public static void main (String arg [])
    {
        String s = JOptionPane.showInputDialog (Null, "Enter a sentence");
        int n = s.indexOf ("CCIT");
        if(n == -1)
            System.out.println ("not found");
        else
            System.out.println ("found at position"+n);
    }
}
```

WAP to read a filename & check if it is an valid .html file or not.

```
class pc
{
    public static void main (String arg [])
    {
        String s = JOptionPane.showInputDialog (Null, "Enter a sentence");
        boolean b = s.endsWith(".html");
        boolean a = s.endsWith(".htm");
        if(b == true || a == true)
            System.out.println("web page");
        Else
            System.out.println("not a web page");
    }
}
```

WAP to separate date, month, & year from a date string .

```
class pc
{
    public static void main (String arg [ ])
    {
        String s = "27-11-2005"
        String p = s.substring (0, 2);
        System.out.println("Date is "+p);
        String x = s.substring (3, 5);
        System.out.println("Month is "+x);
        String z = s.substring (6);
        System.out.println("Year is "+z);
    }
}
```

class Math

This class provide us different mathematical function.
All member of this class are static.

Data member

- 1) **public static final double PI = 3.14**

Eg:- `a = Math.pi * r * r ;`

Method

- 1) **public static int abs (int n)**

It returns absolute value

Eg. `Z = Math.abs (5) + Math.abs (-5)`

- 2) **public static long abs (long n)**

- 3) **public static float abs (float n)**

- 4) **public static double abs (double n)**

- 5) **public static int max (int a, int b)**

Maximum of two numbers.

Eg:- `z = max (4117)`

- 6) **public static int min (int a, int b)**

Will return lowest of two value

Eg:- `z = min (4117, 308);`

Similar method for long, float & double.

- 7) **public static double sqrt (double n)**

It returns square root of given numbers.

Eg:- `double z = Math.sqrt (10);`

- 8) **public static double pow (double a, double b)**

It return **a** to the power **b**.

- 9) **public static double sign (double n)**

Sing of given value.

- 10) **public static double tan (double n)**

- 11) **public static double cos (double n)**

- 12) **public static double ceil (double n)**

It returns lowest whole no which is greater than or equal to given no.

- 13) **public static double floor (double n)**

It returns gretest whole no which is less than or equal to given no.

WAP to print all no. from 1 to 10 along with their square, cube & square root.

```
class pc
{
    public static void main (String args [])
    {
        for (int I =1;I <= 10;I ++)
        {
            System.out.println ("The no. is "+ I );
            double s = Math.pow (I ,2);
            double p = Math.pow (I ,3);
            double q = Math.sqrt (I);
        }
        System.out.println ("Square is =" + s);
        System.out.println ("Cube is =" + p);
        System.out.println ("Square root is =" + q);
    }
}
```

StringBuffer Class

The `java.lang.StringBuffer` class is a growable buffer for Characters. We can create a `StringBuffer` object and its `append()` method:

```
StringBuffer ball = new StringBuffer("Hello");
ball.append(" there.");
ball.append(" How are you?");
```

Constructors

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

methods

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The `append()` method is overloaded like `append(char)`, `append(boolean)`, `append(int)`, `append(float)`, `append(double)` etc.
2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The `insert()` method is overloaded like `insert(int, char)`, `insert(int, boolean)`, `insert(int, int)`, `insert(int, float)`, `insert(int, double)` etc.
3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified `startIndex` and `endIndex`.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified `startIndex` and `endIndex`.

5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

String Vs StringBuffer

| String | StringBuffer |
|--|--|
| String class is immutable. | StringBuffer class is mutable. |
| String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you concat strings. |
| String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

class Vector

Vector implements a dynamic array. Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Vector()

This constructor creates a default vector, which has an initial size of 10

Vector(int size)

This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:

Vector(int size, int incr)

This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward

Vector(Collection c)

creates a vector that contains the elements of collection c

Methods

void add(int index, Object element)

Inserts the specified element at the specified position in this Vector.

boolean add(Object o)

Appends the specified element to the end of this Vector.

boolean addAll(Collection c)

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

void addElement(Object obj)

Adds the specified component to the end of this vector, increasing its size by one.

int capacity()

Returns the current capacity of this vector.

void clear()

Removes all of the elements from this Vector.

Object elementAt(int index)

Returns the component at the specified index.

Object firstElement()

Returns the first component (the item at index 0) of this vector.

Object get(int index)

Returns the element at the specified position in this Vector.

int indexOf(Object elem)

Searches for the first occurrence of the given argument, testing for equality using the equals method.

int indexOf(Object elem, int index)

Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the equals method.

void insertElementAt(Object obj, int index)

Inserts the specified object as a component in this vector at the specified index.

boolean isEmpty()

Tests if this vector has no components.

Object lastElement()

Returns the last component of the vector.

int lastIndexOf(Object elem)

Returns the index of the last occurrence of the specified object in this vector.

int lastIndexOf(Object elem, int index)

Searches backwards for the specified object, starting from the specified index, and returns an index to it.

Object remove(int index)

Removes the element at the specified position in this Vector.

boolean remove(Object o)

Removes the first occurrence of the specified element in this Vector. If the Vector does not contain the element, it is unchanged.

boolean removeAll(Collection c)

Removes from this Vector all of its elements that are contained in the specified Collection.

void removeAllElements()

Removes all components from this vector and sets its size to zero.

boolean removeElement(Object obj)

Removes the first (lowest-indexed) occurrence of the argument from this vector.

void removeElementAt(int index)

removeElementAt(int index)

Object set(int index, Object element)

Replaces the element at the specified position in this Vector with the specified element.

void setElementAt(Object obj, int index)

Sets the component at the specified index of this vector to be the specified object.

int size()

Returns the number of components in this vector.

List subList(int fromIndex, int toIndex)

Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.

void trimToSize()

Trims the capacity of this vector to be the vector's current size.

```
import java.util.Vector;
public class VectorDemo {
    public static void main(String[] args) {
        // create an empty Vector vec with an initial capacity of 4
        Vector vec = new Vector(4);
        // use add() method to add elements in the vector
        vec.add(4);
        vec.add(3);
        vec.add(2);
        vec.add(3);
        // let us get the index of 3
        System.out.println("Index of 3 is :"+vec.indexOf(3));
    }
}
```

Differences between Vector and Array

- Vector is a growable and shrinkable where as Array is not.
- Vector implements the List interface where as array is a primitive data type
- Vector is synchronized where as array is not.
- The size of the array is established when the array is created. As the Vector is growable, the size changes when it grows.

Advantages and Disadvantages of Vector and Array:

- Arrays provide efficient access to any element and can not modify or increase the size of the array.
- Vector is efficient in insertion, deletion and to increase the size.
- Arrays size is fixed where as Vector size can increase.
- Elements in the array can not be deleted, where as a Vector can.

ArrayList Class

ArrayList supports dynamic arrays that can grow as needed. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Constructors and Description

ArrayList()

This constructor builds an empty array list.

ArrayList(Collection c)

This constructor builds an array list that is initialized with the elements of the collection c.

ArrayList(int capacity)

This constructor builds an array list that has the specified initial capacity.

Methods with Description

void add(int index, Object element)

Inserts the specified element at the specified position index in this list. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index > size()`).

boolean add(Object o)

Appends the specified element to the end of this list

boolean addAll(Collection c)

Appends all of the elements in the specified collection to the end of this list.

void clear()

Removes all of the elements from this list.

boolean contains(Object o)

Returns true if this list contains the specified element.

Object get(int index)

Returns the element at the specified position in this list. Throws `IndexOutOfBoundsException` if the specified index is out of range (`index < 0 || index >= size()`).

int indexOf(Object o)

Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

int lastIndexOf(Object o)

Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

Object remove(int index)

Removes the element at the specified position in this list.

Object set(int index, Object element)

Replaces the element at the specified position in this list with the specified element.

int size()

Returns the number of elements in this list.

Object[] toArray()

Returns an array containing all of the elements in this list in the correct order. Throws `NullPointerException` if the specified array is null.

```

import java.util.*;
class demo
{public static void main(String args[ ])
    {
        ArrayList arr=new ArrayList();
        arr.add("Ram");
        arr.add("Ramesh");
        arr.add("Rajesh");
        System.out.println(arr);
    }
}

```

ArrayList Vs Vector

| ArrayList | Vector |
|--|---|
| 1) ArrayList is not synchronized . | Vector is synchronized . |
| 2) ArrayList increments 50% of current array size if number of element exceeds from its capacity. | Vector increments 100% means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is not a legacy class, it is introduced in JDK 1.2. | Vector is a legacy class. |
| 4) ArrayList is fast because it is non-synchronized. | Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses Iterator interface to traverse the elements. | Vector uses Enumeration interface to traverse the elements. But it can use Iterator also. |

Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

| Methods with Description |
|--|
| add() Adds an object to the collection |
| clear() Removes all objects from the collection |
| contains() Returns true if a specified object is an element within the collection |
| isEmpty() Returns true if the collection has no elements |
| remove() Removes a specified object from the collection |
| size() Returns the number of elements in the collection |

Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet


```
import java.util.*;

public class SetDemo {

    public static void main(String args[]) {

        int count[] = {34, 22,10,60,30,22};

        Set<Integer> set = new HashSet<Integer>();

        try{

            for(int i = 0; i<5; i++){

                set.add(count[i]);

            }

            System.out.println(set);

            TreeSet sortedSet = new TreeSet<Integer>(set);

            System.out.println("The sorted list is:");

            System.out.println(sortedSet);

            System.out.println("The First element of the set is: "+

                               (Integer)sortedSet.first());

            System.out.println("The last element of the set is: "+

                               (Integer)sortedSet.last());

        }

        catch(Exception e){}

    }

}
```

HashSet Class

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.

A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.

The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Constructors and Description

HashSet()

This constructor constructs a default HashSet.

HashSet(Collection c)

This constructor initializes the hash set by using the elements of the collection c.

HashSet(int capacity)

This constructor initializes the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.

HashSet(int capacity, float fillRatio)

This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward.

Methods with Description**boolean add(Object o)**

Adds the specified element to this set if it is not already present.

void clear()

Removes all of the elements from this set.

Object clone()

Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.

boolean contains(Object o)

Returns true if this set contains the specified element

boolean isEmpty()

Returns true if this set contains no elements.

Iterator iterator()

Returns an iterator over the elements in this set.

boolean remove(Object o)

Removes the specified element from this set if it is present.

int size()

Returns the number of elements in this set (its cardinality).

TreeSet Class

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

Constructors with Description

TreeSet()

This constructor constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.

TreeSet(Collection c)

This constructor builds a tree set that contains the elements of the collection c.

TreeSet(Comparator comp)

This constructor constructs an empty tree set that will be sorted according to the given comparator.

TreeSet(SortedSet ss)

This constructor builds a TreeSet that contains the elements of the given SortedSet

Methods with Description

void add(Object o)

Adds the specified element to this set if it is not already present.

boolean addAll(Collection c)

Adds all of the elements in the specified collection to this set.

void clear()

Removes all of the elements from this set.

boolean contains(Object o)

Returns true if this set contains the specified element.

Object first()

Returns the first (lowest) element currently in this sorted set.

boolean isEmpty()

Returns true if this set contains no elements.

Iterator iterator()

Returns an iterator over the elements in this set.

Object last()

Returns the last (highest) element currently in this sorted set.

boolean remove(Object o)

Removes the specified element from this set if it is present.

Map Interface

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

Map has its implementation in various classes like HashMap.

| Method | Description |
|--|--|
| <code>public Object put(Object key, Object value)</code> | is used to insert an entry in this map. |
| <code>public void putAll(Map map)</code> | is used to insert the specified map in this map. |
| <code>public Object remove(Object key)</code> | is used to delete an entry for the specified key. |
| <code>public Object get(Object key)</code> | is used to return the value for the specified key. |
| <code>public boolean containsKey(Object key)</code> | is used to search the specified key from this map. |
| <code>public Set keySet()</code> | returns the Set view containing all the keys. |
| <code>public Set entrySet()</code> | returns the Set view containing all the keys and values. |

HashMap Class

The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get() and put(), to remain constant even for large sets.

Constructors and Description

HashMap()

This constructor constructs a default HashMap.

HashMap(Map m)

This constructor initializes the hash map by using the elements of the given Map object m

HashMap(int capacity)

This constructor initializes the capacity of the hash map to the given integer value, capacity.

HashMap(int capacity, float fillRatio)

This constructor initializes both the capacity and fill ratio of the hash map by using its arguments.

```
import java.util.*;
public class CollectionsDemo {
    public static void main(String[] args) {
        Map m1 = new HashMap();
        m1.put("Zara", "8");
        m1.put("Mahnaz", "31");
        m1.put("Ayan", "12");
        System.out.println(" Map Elements");
        System.out.print("\t" + m1);
    }
}
```

Clas Date

The java.util.Date class represents a specific instant in time, with millisecond precision.

Constructor

Date() :- create a Date Object containing current Date

Date(long date) :- create a Date Object and initializes it to represent the specified number of milliseconds since January 1, 1970, 00:00:00 GMT.

Methods

boolean after(Date when)

This method tests if this date is after the specified date.

boolean before(Date when)

This method tests if this date is before the specified date.

long getTime()

This method returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.

void setTime(long time)

This method sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT.

String toString()

This method converts this Date object to a String of the form.

Int getDate()

returns date from date object

Class Calendar

The **java.util.calendar** class provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.

Method

abstract void add(int field, int amount)

This method adds or subtracts the specified amount of time to the given calendar field, based on the calendar's rules.

boolean after(Object when)

This method returns whether this Calendar represents a time after the time represented by the specified Object.

boolean before(Object when)

This method returns whether this Calendar represents a time before the time represented by the specified Object.

int get(int field)

This method returns the value of the given calendar field.

Date getTime()

This method returns a Date object representing this Calendar's time value (millisecond offset from the Epoch").

void set(int year, int month, int date)

This method sets the values for the calendar fields YEAR, MONTH, and DAY_OF_MONTH.

String toString()

This method return a string representation of this calendar.

static Calendar getInstance()

This method gets a calendar using current date & time

```
import java.util.Calendar;

public class CalendarDemo {

    public static void main(String[] args) {

        // create a calendar

        Calendar cal = Calendar.getInstance()

        // print current date

        System.out.println("The current date is : " + cal.getTime());

        // add 20 days to the calendar

        cal.add(Calendar.DATE, 20);

        System.out.println("20 days later: " + cal.getTime());

        // subtract 2 months from the calendar

        cal.add(Calendar.MONTH, -2);

        System.out.println("2 months ago: " + cal.getTime());

        // subtract 5 year from the calendar

        cal.add(Calendar.YEAR, -5);

        System.out.println("5 years ago: " + cal.getTime());

    }

}
```

Scanner class

The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression.

| Method | Description |
|----------------------------|---|
| public String next() | it returns the next token from the scanner. |
| public String nextLine() | it moves the scanner position to the next line and returns the value as a string. |
| public byte nextByte() | it scans the next token as a byte. |
| public short nextShort() | it scans the next token as a short value. |
| public int nextInt() | it scans the next token as an int value. |
| public long nextLong() | it scans the next token as a long value. |
| public float nextFloat() | it scans the next token as a float value. |
| public double nextDouble() | it scans the next token as a double value. |

Java Modifiers

| Modifier | Used on | Meaning |
|---------------|-----------|--|
| abstract | Class | Contains unimplemented methods and cannot be instantiated. |
| | interface | All interfaces are abstract. Optional in declarations |
| | method | No body, only signature. The enclosing class is abstract |
| final | Class | Cannot be subclassed |
| | method | Cannot be overridden and dynamically looked up |
| | field | Cannot change its value. static final fields are compile-time constants. |
| | variable | Cannot change its value. |
| native | method | Platform-dependent. No body, only signature |
| none(package) | Class | Accessible only in its package |
| | interface | Accessible only in its package |
| | member | Accessible only in its package |
| private | member | Accessible only in its class(which defines it). |
| protected | member | Accessible only within its package and its subclasses |

| Modifier | Used on | Meaning |
|--------------|---|---|
| public | class interface member | Accessible anywhere Accessible anywhere Accessible anywhere its class is. |
| static | class method field initializer | Make an inner class top-level class A class method, invoked through the class name. A class field, invoked through the class name one instance, regardless of class instances created. Run when the class is loaded, rather than when an instance is created. |
| synchronized | method | For a static method, a lock for the class is acquired before executing the method. For a non-static method, a lock for the specific object instance is acquired. |
| transient | field | Not be serialized with the object, used with object serializations. |
| volatile | field | Accessible by unsynchronized threads, very rarely used. |
| strictfp | classes interfaces methods | It restricts floating-point calculations to ensure portability. As different platforms can handle, different floating-point calculations precisions and greater range of values than the java specification requires, which may produce different output on different platforms. |

Applications Vs Applets

| Applets | Applications |
|---|---|
| Applet does not use main() method for initiating execution of code | Application use main() method for initiating execution of code |
| Applet cannot run independently | Application can run independently |
| Applet cannot read from or write to files in local computer | Application can read from or write to files in local computer |
| Applet cannot run any program from local computer. | Application can run any program from local computer. |
| Applet are restricted from using libraries from other language such as C or C++ | Application are not restricted from using libraries from other language |

Applets

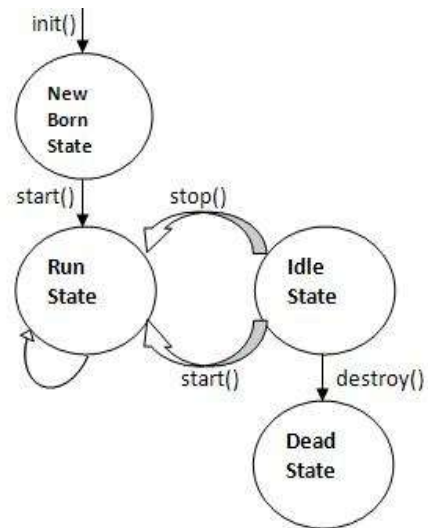
An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the `java.applet.Applet` class.
- A `main()` method is not invoked on an applet, and an applet class will not define `main()`.
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle Methods of Applet

Five methods in the Applet class give you the framework on which you build any serious applet:



- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

Life Cycle states Of An Applet

These are the different stages involved in the life cycle of an applet:

Initialization State : This state is the first state of the applet life cycle. An applet is created by the method `init()`. This method initializes the created applet. It is Called exactly once in an applet's life when applet is first loaded, which is after object creation, e.g. when the browser visits the web page for the first time.Used to read applet parameters, start downloading any other images or media files, etc. `Init()` method should be overridden in our applet.

Running State : This state is the second stage of the applet life cycle. This state comes when `start()` method is called. Called at least once.Called when an applet is started or restarted, i.e., whenever the browser visits the web page.

Idle or Stopped State : This state is the third stage of the applet life cycle. This state comes when `stop()` method is called implicitly or [explicitly](#).`stop()` method is called implicitly. It is called at least once when the browser leaves the web page when we move from one page to another. This method is called only in the running state and can be called any number of times. It should be overridden in our applet.

Dead State : This state is the fourth stage of the applet life cycle. This state comes when `destroy()` method is called. In this state the applet is completely removed from the memory. It called exactly once when the browser unloads the applet.Used to perform any final clean-up. It occurs only once in the life cycle. It should be overridden in our applet.

Display State : This state is the fifth stage of the applet life cycle. This state comes when use s the applet displays something on the screen. This is achieved by calling `paint()` method.`Paint()` method can be called any number of times.This can be called only in the running state.This method is a must in all applets. It should be overridden in our applet.

Applet Tag in a Web Page

Web pages are written in a language called HTML. This explanation of how to embed an applet in a Web page assumes that you have some knowledge of basic HTML. An applet is embedded in a Web page using an `<applet>` tag. A minimal `<applet>` tag looks as follows:

```
<applet code=Clock height=300 width=350>  
</applet>
```

The code attribute of this sample `<applet>` tag specifies that the applet to be run is a class named `Clock`. The width and height attributes specify that the applet should be given a screen area that is 300 pixels high and 350 pixels wide.

The following list shows all of the attributes that can be specified in an `<applet>` tag. The attributes should be specified in the order in which they are listed. The code, height, and width attributes are required in an `<applet>` tag; the other attributes are optional:

codebase

The codebase attribute should specify a URL that identifies the directory used to find the `.class` files needed for the applet

code

The code attribute specifies the name of the class that implements the applet.

alt

The alt attribute specifies the text that should be displayed by Web browsers that understand the `<applet>` tag but cannot run Java applets.

width

The width attribute specifies the width of the applet in pixels.

height

The height attribute specifies the height of the applet in pixels.

Applet Execution

From the viewpoint of the host application, the interaction typically follows a standard sequence of events. The host application usually does the following:

1. Installs a `SecurityManager` object to implement a security policy.
2. Creates a `ClassLoader` object to load the applet.
3. Loads the applet and calls its default constructor.
4. Passes an `AppletStub` object to the applet's `setStub()` method.
5. Calls the applet's `init()` method in a separate thread.
6. Marks the applet as active.
7. Starts a new thread to run the applet's `start()` method.
8. Calls the applet's `show()` method, which makes the applet visible and causes the applet's `paint()` method to be called for the first time.
9. Calls the applet's `paint()` method whenever the applet needs to be refreshed.
10. Calls the applet's `start()` and `stop()` methods when the host wants the applet to start or stop. These methods are typically called when the applet is exposed or hidden.

Calls the applet's `hide()` method followed by its `destroy()` method when the host wants to shut down the applet.

Applet Parameters

Many times when you're developing a Java applet, you want to pass parameters from an HTML page to the applet you're invoking. For instance, you may want to tell the applet what background color it should use, or what font to use etc.

A great benefit of passing applet parameters from HTML pages to the applets they call is portability. If you pass parameters into your Java applets, they can be portable from one web page to another, and from one web site to another.

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named `getParameter()`.
Syntax:

public String getParameter(String parameterName)

for eg:

```
import java.applet.Applet;
import java.awt.Graphics;
public class UseParam extends Applet
{
    public void paint(Graphics g){
        String str=getParameter("msg");
        g.drawString(str,50, 50);
    }
}
```

myapplet.html

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```

Class Applet

This is a base class from which all applet classes are derived. This class provides us different features required to build an applet.

Browser called its different method on different event.

Methods–

public void paint(Graphics g) -

This method is called by the browser whenever it wants to show applet. We can override this method if we want to perform some drawing and painting operation.

public void init() -

This is the first method which is called by the browser after creating the applet object. We can override this method, if we want to perform some initialization.

public void start() -

This method called by the browser when applets scroll into view.

public void stop() -

This method is called by browser whenever an applet scroll out of the view.

public void destroy() -

This method called by the browser before removing the applet.

public void showStatus(String msg) -

We can call this method to show some message into browser statusbar.

public URL getDocumentBase() -

It returns the URL of the document /site from where the document is received.

public URL getCodeBase() -

It returns the URL of site from where the code is received.

public String getParameter(String parname) -

It returns the parameter value of given parameter name.

public Image getImage(String filename) -

It returns a image object from specified resource.

Class Graphics

This class provide us different method for drawing and painting.

| Method | Description |
|--|--|
| <code>void drawRect(int x,int y,int w ,int h)</code> | Will draw a rectangle. |
| <code>void fillRect(int x, int y, int w, int h)</code> | Will draw a fillrectangle. |
| <code>void drawOval(int x,inty,int w,int h)</code> | It will draw oval. |
| <code>void fillOval(int x,int y, int w,int h)</code> | It will draw filloval. |
| <code>void clearRect(int x,int y,int w,int h)</code> | It will clear the specified rectangular area. |
| <code>void drawLine(int x,int y,int x1,int y1)</code> | It draw line between two point . |
| <code>void drawArc(int x,int y,int w,int h,int start angle,int endangle)</code> | It will draw arc. |
| <code>void fillArc(int x,int y,int w,int h,int start angle,int endangle)</code> | It will draw fill arc. |
| <code>void setColor(Color c)</code> | Set the color. |
| <code>void setFont(Font f)</code> | Set the font. |
| <code>void drawstring(String s,int x,int y)</code> | Will display string at given position. |
| <code>void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)</code> | Draws a closed polygon |
| <code>void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)</code> | Fills a closed polygon |
| <code>boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)</code> | Draws as much of the specified image as has already been scaled to fit inside the specified rectangle. |

```
//program to draw ten concentric circles.
import java.applet.*;
import java.awt.*;
public class g1 extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        for(int i=0,j=100;i<=50;i=i+5,j=j-10)
            g.drawOval(100-i,100-i,100-j,100-j);
    }
}
```

AWT components

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.

AWT Container

A container is an object in which we can add awt components . The job of the container is to arrange the components which are added into the container . To arrange the components it has a `LayoutManager` object .AWT provides us different container by calling its method

void add(Component c)

AppletViewer

AppletViewer is a standalone command-line program from Sun to run Java applets.Appletviewer is generally used by developers for testing their applets before deploying them to a website. As a Java developer, it is a preferred option for running Javaapplets that do not involve the use of a web browser.

Class Component

The Component class defines a number of methods that can be used on any of the classes that are derived from it. Methods listed below can be used on all User Interface(UI) components as well as containers.

| Method | Description |
|---|---|
| void setSize(int width , int height) | Resizes the corresponding component so that it has width and height. |
| void setFont(Font f) | Sets the font of the corresponding component. |
| void setEnabled(boolean b) | Enables or disables the corresponding component ,depending on the value of the parameter b. |
| void setVisible (boolean b) | Shows or hides the corresponding component depending on the value of parameter b. |
| void setForeground(Color c) | Sets the foreground color of the corresponding component. |
| void setBackground(Color c) | Sets the background color of the corresponding component. |
| void setBounds(int x,int y,int w,int h) | resizes the component. |
| Color getBackground() | Gets the background color of the corresponding component. |
| Color getForeground() | Gets the foreground color of the corresponding component. |
| Dimention getBounds() | Gets the bounds of the corresponding component . |
| Dimention getSize() | Returns the size of the component in the form of a Dimension object. |
| Font getFont() | Gets the font of the corresponding component. |

Event Delegation Model

This model is based on the concept of an event listener. An object interested in receiving events is an event listener. An event is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated. The Delegation Event Model has the following key participants

- **Event** - Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button etc.
- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler.
- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

In an AWT program, the event source is typically a GUI component and the listener is commonly an "adapter" object which implements the appropriate listener (or set of listeners) in order for an application to control the flow/handling of events. The listener object could also be another AWT component which implements one or more listener interfaces for the purpose of hooking GUI objects up to each other.

class Button

Buttons are components that can contain a label. Button is similar to a push button in any other GUI. Pushing a button causes the run time system to generate an event. This event is sent to the program. The program in turn can detect the event and respond to the event. Clicking the button generate the ActionEvent.

| Constructor | Description |
|----------------------|---|
| Button() | Constructs a button with no label. |
| Button(String label) | Constructs a button with the label specified. |

| | |
|---|--|
| String getLabel() | Returns the label of the corresponding button. |
| void setLabel(String label) | Sets label of button to the value specified. |
| void addActionListener(ActionListener L) | Adds specified action listener to receive action events from corresponding button. |

interface ActionListener

This interface is implemented to listen events of action component such as Button ,Menu Item.

Method –

public void actionPerformed(ActionEvent e) –

This method is called of a listener when a button or menu item is clicked.

class ActionEvent

An object of this type contain the information about the source which has generated the event .

| | |
|------------------------------|--|
| String getActionCommand() | Returns the label of the button which has generated the event . |
| Object getSource() | It returns the reference of the button which has generated the event . |

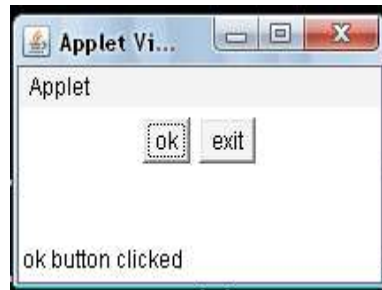
//program to check which button is clicked using getSource() method.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class MyApplet extends Applet implements ActionListener
{
    Button b1,b2;
    public void init()
    {
        b1= new Button("ok");
        b2= new Button("exit");
        add(b1);
        add(b2);
        b1.addActionListener(this);
        b2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        Button bs=(Button)e.getSource();
        if(bs==b1)
        {
            showStatus("ok button clicked");
        }
        if(bs==b2)
        {
            showStatus("exit button clicked");
        }
    }
}
```



//program to check which button is clicked using getActionCommand() method.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class MyApplet extends Applet implements ActionListener
{
    Button b1,b2;
    public void init()
    {
        b1= new Button("ok");
        b2= new Button("exit");
        add(b1);
        add(b2);
        b1.addActionListener(this);
        b2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String s1=e.getActionCommand();
        if(s1.equals("ok"))
        {
            showStatus("ok button clicked");
        }
        if(s1.equals("exit"))
        {
            showStatus("exit button clicked");
        }
    }
}
```



Class Label

Labels do not generate any event .

This component can be used for displaying a single line of text in a container.

| Constructor | Description |
|------------------------------------|--|
| Label() | Constructs an empty Label. |
| Label(String text) | Constructs a string with corresponding text. |
| Label(String text , int alignment) | Constructs a string with the text, with the specified alignment .The alignment can be CENTER,LEFT,RIGHT. |

Methods –

| | |
|---------------------------|--|
| String getText() | Gets the text of the corresponding label. |
| void setText(String text) | Sets the text for the corresponding label to the specified text. |

Class Font

This class is used to create Font object and we can set Fonts using method

setFont(Font f) of class Component.

| Constructor | Description |
|---|--|
| Font (String fontname,int style,int size) | Create Font object with given fontname,style and size. |

Fontname :

TimesRoman,MonoSpaced,Dialog,DialogInput,ScanScript,Courier etc.

Styles :

```
public static final int BOLD;
public static final int ITALIC;
public static final int PLAIN;
```

String getFamily()

Returns the family name of this Font.

String getFontName()

Returns the font face name of this Font.

int getSize()

Returns the point size of this Font, rounded to an integer.

boolean isBold()

Indicates whether or not this Font object's style is BOLD.

boolean isItalic()

Indicates whether or not this Font object's style is ITALIC.

Code to set font for a button

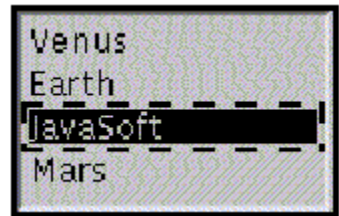
```
Eg. Font f=new Font("TimesRoman",Font.BOLD,100);
    b.setFont(f);
```

Class List

List components present the user with a scrolling list of text items.

The differences between a list and a choice menu are:

Unlike Choice, which displays only the single selected item, the list can be made to show any number of choices in the visible window.



The list can be constructed to allow multiple selections.

| Constructor | Description |
|---|--|
| List() | Creates a new scrolling list of items. |
| List(int visibleitems) | Creates a new scrolling list of items with the specified number of visible items. |
| List(int visibleitems, boolean multimode) | Creates a new scrolling list of items to display the specified number of visibleitems. |

| Method | Description |
|---------------------------------|---|
| void add(String item) | Add the specified item to the end of the scrolling list |
| void add(String item,int index) | Adds the specified item at the position |
| String getItem(int index) | Gets the item at the specified index. |
| int getItemCount() | Gets the no. of items in the list. |
| String [] getItems() | Gets the items in the list. |
| int getSelectedIndex() | Gets the index of the selected item in the list |
| String getSelectedItem() | Gets the selected item in the corresponding list |
| void setMultipleMode(boolean b) | Sets the flag that allows multiple selection |

| | |
|---|--|
| <code>void select(int index)</code> | Selects the item at the specified index |
| <code>void remove(int index)</code> | Will remove the item at specified index. |
| <code>void remove(String item)</code> | Will remove specified item. |
| <code>void removeAll()</code> | Remove all the items in the list. |
| <code>void addItemListener(Listener L)</code> | This method tells the list control to call a special method <code>itemStateChanged(ItemEvent e)</code> when item selected in list. |

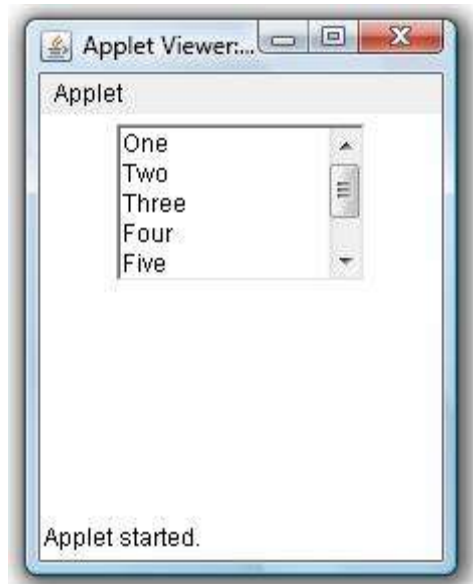
```
import java.applet.Applet;
import java.awt.List;
```

```
public class CreateListExample extends Applet
```

```
{
    public void init()
    {
        //this list will have 5 visible rows
        List list = new List(5);

        list.add("One");
        list.add("Two");
        list.add("Three");
        list.add("Four");
        list.add("Five");
        list.add("Six");
        list.add("Seven");

        //add list
        add(list);
    }
}
```



Class Choice

The Choice class implements a pop up menu that allows the user to select an item from that menu .This UI component displays the currently selected item with an arrow to its right. On clicking the arrow, the menu opens and displays options for a user to select.

| Constructor | Description |
|-------------|----------------------------|
| Choice() | Creates a new choice menu. |

| Methods | Description |
|-------------------------------|---|
| void add(String item) | Adds an item to the corresponding choice menu. |
| void addItem(String item) | Adds an item to the corresponding choice . |
| String getItem(int index) | Gets the string at the specified index of the corresponding choice menu. |
| int getItemCount() | Returns the no. of items in the corresponding choice menu. |
| int getSelectedIndex() | Returns the index of the currently selected item. |
| String getSelectedItem() | Gets a representation of the current choice as a string. |
| insert(String item,int index) | Inserts the item into the corresponding choice at the specified position. |
| void remove(int position) | Removes the first occurrence of item from the corresponding Choice menu. |
| void removeAll() | Removes all the items from the corresponding Choice menu. |
| select(int position) | Sets the selected item in the corresponding Choice menu to be the item at specified position. |
| select(String str) | Sets the selected item in the corresponding Choice menu to be the item whose name is equal to the specified string. |

Interface ItemListener

This interface is used to listen action of the List ,Choice, Checkboxes, Radiobuttons.

Method –

public void itemStateChanged(ItemEvent e)-

* Program to display different shapes according to different choices.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class pcListApplet extends Applet
implements ItemListener
{ Choice ch;
  public void init()
  {   ch=new Choice();
      add(ch);
      ch.addItemListener(this);
      ch.addItem("Circle");
      ch.addItem("Rect");
      ch.addItem("Pie");
  }
  public void itemStateChanged(ItemEvent e)
  {   Graphics g=this.getGraphics();
      String item=ch.getSelectedItem();
      g.clearRect(50,50,100,100);
      if(item.equals("Circle"))
          g.fillOval(50,50,100,100);
      if(item.equals("Rect"))
          g.fillRect(50,50,100,100);
      if(item.equals("Pie"))
          g.fillArc(50,50,100,100,0,90);
  }
}
```

Circle ▾



Class TextField

TextFields are UI components that accept text input from the user. TextFields only have single line text .

| Constructor | Description |
|------------------------------------|---|
| TextField() | Constructs a new text field |
| TextField(int columns) | Creates a new text field with the specified number of columns. |
| TextField(String text) | Constructs a new text field initialized with the specified text. |
| TextField(String text,int columns) | Constructs a new text field initialized with the specified text with the specified number of columns. |

| Methods | Description |
|------------------------------|--|
| int getColumns() | Gets the no. of columns in the corresponding text field. |
| char getEchoChar() | Gets the character that is to be used for echoing. |
| void setColumns(int columns) | Sets the no. of columns in the text field. |
| void setEchoChar(char c) | Sets the echo character for the text field. |
| void setText(String str) | Sets the text |

class TextComponent

TextComponent is a base class for class TextField & TextArea.

| Methods | Description |
|---------------------------------|--|
| void setText(String str) | sets the given string into the textbox |
| String getText() | Gets the text from textbox. |
| void select(int start, int end) | Selects the text in specified range |
| String getSelectedText() | Gets the selected text from textbox |
| int getSelectionStart() | Gets starting position of selected Text. |
| int getSelectionEnd() | Gets ending position of selected Text. |

// program to exchange contents of 2 textboxes

| | | |
|------|----------|----------|
| CCIT | Exchange | Amravati |
|------|----------|----------|

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ExgApplet extends Applet implements ActionListener
{
    TextField t1,t2;
    Button b;
    public void init()
    {
        t1=new TextField(10);
        add(t1);
        b=new Button("Exchange");
        add(b);
        b.addActionListener(this);
        t2=new TextField(10);
        add(t2);
    }
    public void actionPerformed(ActionEvent e)
    {
        String temp=t1.getText();
        t1.setText(t2.getText());
        t2.setText(temp);
    }
}
```

class TextArea

TextAreas behave like TextFields except that they have more functionality to handle large amounts of text .

- TextArea can contain multiple rows of text.
- TextArea have scrolls that permit handling of large amounts of data.

| Constructors | Description |
|---|---|
| TextArea() | Constructs new text area. |
| TextArea(int rows,int columns) | Constructs new text area with specified no. of rows and columns. |
| TextArea(String text) | Constructs new text area with the specified text. |
| TextArea(String text, int rows,int columns) | Constructs new text area with specified text and specified no. of rows and columns. |
| TextArea(String text, int rows,int columns,int scrollbar) | Constructs new text area with specified text and specified no. of rows and columns and visibility of the scrollbar. |

| Methods | Description |
|---------------------------------|---|
| void append(String str) | Appends the given string to the text area's content. |
| int getColumns() | Gets the no. of columns in the text area. |
| int getRows() | Gets the no. of rows in the text area. |
| void insert(String str,int pos) | Inserts the specified string at the specified position. |
| void setColumns(int columns) | Sets the no. of columns for the text area |
| void setRows(int rows) | Sets the no. of rows for the text area |

Class Color

An object of this type represent Color.

This provide us different readymade Color object which we can use.

```
public static final Color red;  
public static final Color blue;  
public static final Color green;  
public static final Color white;  
public static final Color black;  
public static final Color yellow;  
public static final Color orange;  
public static final Color pink;  
public static final Color magenta;  
public static final Color cyan;  
public static final Color lightGray;  
public static final Color darkGray;
```

Constructor -

Color (int redcolor,int greencolor,int bluecolor)-

It create color object by intensity of red, green, blue . Intensity must be in range 0 to 255.

Code to set color of a Button

Eg.

```
b.setColor(Color.red);
```

or

```
Color c=new Color(150,150,180);  
b.setColor(c);
```

Class Checkbox

This class is used to create checkbox type object.

| Constructor | Description |
|---|--|
| Checkbox() | It is used to create Checkbox without label. |
| Checkbox(String label) | It is used to create Checkbox with specified label. |
| Checkbox(String label, boolean state) | It is used to create Checkbox with specified label and state.(state can be true or false). |
| Checkbox((String label, CheckboxGroup g,boolean state) Checkbox((String label, boolean state, CheckboxGroup g) | Both the constructor is used to create radiobuttons. |

| Methods | Description |
|---|--|
| void setLabel(String label) | It is used to set label on Checkbox. |
| String getLabel() | It returns the label of the Checkbox. |
| boolean getState() | It returns the state of the Checkbox. |
| setState(boolean b) | It is used to set state of Checkbox. |
| public void addItemListener(ItemListener l) | This method tells the Checkbox to call a method "itemStateChanged" when checkbox state is changed. |

Class CheckboxGroup

This class is used to group radiobutton and user can select only one radiobutton in group.

| Constructor | Description |
|-----------------|---------------------------------------|
| CheckboxGroup() | It is used to create Checkbox object. |

| Methods | Description |
|------------------------------------|---|
| Checkbox getSelectedCheckbox()- | Returns the reference of the checkbox object. |

//Program to Calculate salary according to selected Checkboxes.

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class pcBonusApplet extends
Applet implements ItemListener
{Label l1,l2;
  TextField t1,t2;
  Checkbox cb1,cb2,cb3;
  Button b;
  public void init()
  {
    l1=new Label("Enter Basic Salary"); add(l1);
    t1=new TextField(10); add(t1);
    cb1=new Checkbox("Add Bonus Rs 2000"); add(cb1);
    cb2=new Checkbox("Add HRA Rs 1000"); add(cb2);
    cb3=new Checkbox("Deduct ITax Rs 5000"); add(cb3);
    b=new Button("Net Salary");
    add(b); b.addActionListener(this);
    t2=new TextField(10); add(t2);
  }
  public void actionPerformed(ActionEvent e)
  {
    int bs=Integer.parseInt(t1.getText());
    if(cb1.getState())
      bs=bs+2000;
    if(cb2.getState())
      bs=bs+1000;
    if(cb3.getState())
      bs=bs-5000;
    t2.setText(String.valueOf(bs));
  }
}

```

Enter Basic Salary

25000

☒ Add Bonus Rs 2000☐ Add HRA Rs 1000☒ Deduct ITax Rs 5000

Net Salary

22000

// program to calculate net salary according to selected RadioButton

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class pcBonusApplet extends Applet implements ItemListener
{
    Label l1,l2;
    TextField t1,t2;
    Checkbox cb1,cb2,cb3;
    CheckboxGroup g;
    public void init()
    {
        l1=new Label("Enter
        Basic Salary"); add(l1);
        t1=new TextField(10);
        add(t1);
        g=new CheckboxGroup();
        cb1=new Checkbox("Add Bonus Rs 2000",g,false);
        add(cb1); cb1.addItemListener(this);
        cb2=new Checkbox("Add Bonus Rs 1000",g,false);
        add(cb2); cb2.addItemListener(this);
        cb3=new Checkbox("No Bonus",g,false);
        add(cb3); cb3.addItemListener(this);
        t2=new TextField(10); add(t2);
    }
    public void itemStateChanged(ItemEvent e)
    {
        int bs=Integer.parseInt(t1.getText());
        Checkbox cs=g.getSelectedCheckbox();
        if(cs==cb1)
            bs=bs+2000;
        if(cs==cb2)
            bs=bs+1000;
        t2.setText(String.valueOf(bs));
    }
}
```

Enter Basic Salary

25000

☒ Add Bonus Rs 2000☐ Add Bonus Rs 1000☐ No Bonus 27000

Class Scrollbar

Scrollbar are used to select a value between a specified minimum and maximum.

The arrows at either end allow incrementing or decrementing the value represented by the scrollbar.

The thumb's position represents the value of the scrollbar.

The various constructors that can be used to create Scrollbar are :

| Constructor | Description |
|---|---|
| Scrollbar() | Constructs a vertical scrollbar. |
| Scrollbar(int orientation) | Constructs a new scrollbar with the specified orientation |
| Scrollbar(int orientation,int maxvalue,int visible,int minimum,int maximum) | Constructs a new scrollbar with the specified orientation ,initial value, page size,minimum and maximum values. |

| Methods | Description |
|---|--|
| int getValue() | Gets the current value of scrollbar. |
| void setBlockIncrement(int v) | Sets the block increment scroll bar. |
| setMaximum(int n) | Sets the maximum value scrollbar. |
| setMinimum(int n) | Sets the minimum scrollbar. |
| setValue(int n) | Sets the value of the scrollbar |
| public void addAdjustmentListener(AdjustmentListener L) | This method tells the scrollbar to call the special method adjustmentValueChanged(AdjustmentEvent e) whe n the scrolling event occurs. |

Interface AdjustmentListener

Methods of this interface is implemented to listen scrolling events.

Methods -

public void adjustmentValueChanged(AdjustmentEvent e) -

This method is called when scrolling event occurs.

Class AdjustmentEvent

Object of this contain information about that object which has generated the scrolling events.

Methods –

| Methods | Description |
|--------------------|--|
| int getValue() | Returns the current thumb position. |
| Object getSource() | Returns the reference of the object which has generated the event. |

//program implementing scrollbar and its method and events.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class xscroll extends
Applet implements AdjustmentListener
{ Scrollbar s1,s2,s3;
public void init()
{s1= new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,255);add(s1);
s2= new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,255);add(s2);
s3= new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,255);add(s3);
s1.addAdjustmentListener(this);
s2.addAdjustmentListener(this);
s3.addAdjustmentListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent e)
{ int r=s1.getValue();
int g=s2.getValue();
int b= s3.getValue();
Color c= new Color (r,g,b);
setBackground (c);
showStatus(String.valueOf(r)+" , "+String.valueOf(g)+" ,
"+String.valueOf(b));
}
}
```



LayoutManagers

Every container has a LayoutManager attached with it . Job of the LayoutManager is to arrange the components which are added in the container .Different LayoutManager are derived from base class LayoutManager.

class FlowLayout

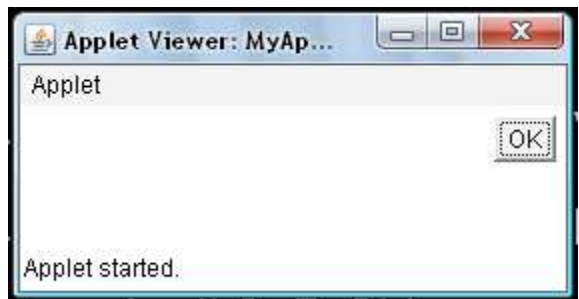
This LayoutManager arranges the component like words on a page .i.e. from left to right and top to bottom. This is the default LayoutManager of the container Applet.

| Constructor | Description |
|---|---|
| FlowLayout() | LayoutManager with default Layout center. |
| FlowLayout(int alignment) | Will create a LayoutManager with specified alignment. |
| FlowLayout(int alignment,int hgap,int vgap) | LayoutManager with specified alignment with specified hgap and vgap between two components. |

Alignment can be. LEFT , RIGHT, CENTER;

// applet containing a button at top right position

```
import java.awt.*;
import java.applet.*;
public class pcFLayout
extends Applet
{ Button b;
  public void init()
  { FlowLayout f= new FlowLayout(FlowLayout.RIGHT);
    setLayout(f);
    b=new Button("OK");
    add(b);
  }
}
```



class GridLayout

This LayoutManager arranges the components in a grid fashion. i.e. in rows and columns.

| Constructor | Description |
|---|--|
| GridLayout(int rows, int columns) | Creates a grid of no. of rows and columns specified. |
| GridLayout(int rows, int columns, int hgap, int vgap) | with specified hgap and vgap |

// applet containg 9 buttons arranged in 3 X 3

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class pcLMGrid extends Applet
implements ActionListener
{ Button b[];
public void init()
{ setLayout(new GridLayout(3,3,5,5));
  b=new Button[9];
  for(int i=0;i<b.length;i++)
  {
    b[i]=new Button(String.valueOf(i+1));
    add(b[i]);
    b[i].addActionListener(this);
  }
}
public void actionPerformed(ActionEvent e)
{ String cmd=e.getActionCommand();
  showStatus(cmd);
}
}
```



class BorderLayout

This LayoutManager arranges the components along the border and at the center of the container. While adding the component we have to specify the constraint where we want to add by calling method

add(Component c, Object constraint) –

constraint can be any of the

"North", "South", "West", "East", "Center".

| Constructor | Description |
|---------------------------------|--------------------------------------|
| BorderLayout() | Arranges component along the borders |
| BorderLayout(int hgap,int vgap) | with specified hgap and vgap |

//applet containing 2 buttons at Top & Bottom

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class pcLMBorder extends
Applet implements ActionListener
{Button b1,b2;
public void init()
    { setLayout(new BorderLayout());
      b1=new Button("OK");
      add(b1,"North");
      b1.addActionListener(this);
      b2=new Button("Cancel");
      add(b2,"South");
      b2.addActionListener(this);
    }
public void actionPerformed(ActionEvent e)
    { String cmd=e.getActionCommand();
      showStatus(cmd);
    }
}
```



class CardLayout

This LayoutManager arranges the components like a dake of cards. It creates each component like as large as container so only one component is visible at a time.

| Constructor | Description |
|-------------------------------|-------------|
| CardLayout() | |
| CardLayout(int hgap,int vgap) | |

| Methods | Description |
|---------------------------------|--|
| void next(Container parent) | It will show next component from container |
| void previous(Container parent) | will show previous component |
| void first(Container parent) | It will show first component in container |
| void last(Container parent) | It will show last component in container |

import java.awt.*; // **Example of LayoutManager CardLayout**

import java.awt.event.*;

import java.applet.*;

public class pcLMCard extends Applet implements ActionListener

{ Button b[];

String s[]={"SUN","MON","TUE","WED","THR","FRI","SAT"};

CardLayout cl;

public void init()

{ cl=new CardLayout();

setLayout(cl);

b=new Button[7];

for(int i=0;i<b.length;i++)

{ b[i]=new Button(s[i]);

add(b[i],"Center");

b[i].addActionListener(this); }

}

public void actionPerformed(ActionEvent e)

{ cl.next(this);

}}



class Panel

Panel is a container in which we can add multiple components. This class is derived from the class Component. So we can add Panel into another container like component.

| Constructor – | Description |
|--------------------------|--|
| Panel() | Create blank Panel object . |
| Panel(LayoutManager lm). | Create Panel object with given layout. |

//Example of Panel

```
public class pcPanelApplet extends
Applet implements ActionListener
{ Button b[ ];
Label lbl;
Panel p;
Color co[ ]={Color.red, Color.orange, Color.yellow, Color.green,
Color.blue };
public void init()
{ setLayout(new BorderLayout());
lbl=new Label("CCIT",Label.CENTER);
add(lbl,"Center");
lbl.setFont(new Font("TimesRoman",Font.BOLD,100));
p=new Panel(new GridLayout(1,5,10,10));
add(p,"South");
b=new Button[5];
for(int i=0;i<b.length;i++)
{ b[i]=new Button(); p.add(b[i]);
b[i].addActionListener(this);
b[i].setBackground(co[i]);
}
}
public void actionPerformed(ActionEvent e)
{ Button bs=(Button)e.getSource();
Color cs=bs.getBackground();
lbl.setForeground(cs);
}}
```



Windows application

We can build GUI application and for this we required different window classes .Java provides us classes like Frame,Dialog.

To create standard window we can use class Frame.

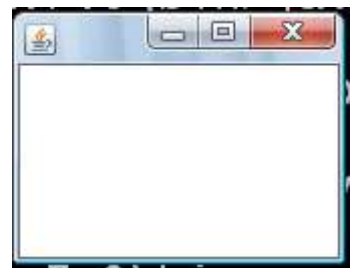
Class Frame –

This is used to create standard application window. Default LayoutManager of the frame object is BorderLayout. To close Frame without using WindowListener we can use a special method of class System. System.exit(0);

| Constructor | Description |
|---------------------|---|
| Frame() | Create a blank Frame object without title. |
| Frame(String title) | Create a Frame object with specified title. |

| Methods | Description |
|------------------------------|--|
| void setTitle(String title) | It is used to set title of Frame object. |
| String getTitle() | It returns the title of the Frame. |
| setMenuBar(MenuBar mb) | It is used to attach MenuBar object. |
| void setResizable(Boolean b) | It creates resizable or fixed window. |
| void setSize(int w,int h) | Sets the size of the window. |
| void setVisible() | It shows or hides the window. |
| | |

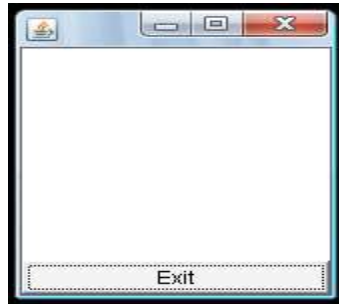
```
import java.awt.*;           //Example of Simple Frame
class pc
{
    public static void main(String args[])
    {
        Frame f=new Frame();
        f.setSize(400,200);
        f.setVisible(true);
    }
}
```



*** Example of Simple Frame with Exit Button**

```
import java.awt.*;  
import java.awt.event.*;
```

```
class myFrame extends Frame implements ActionListener  
{  
    Button b;  
    public myFrame()  
    {  
        b=new Button("Exit");  
        add(b,"South");  
        b.addActionListener(this);  
    }  
}
```



```
    public void actionPerformed(ActionEvent e)  
    {  
        System.exit(0);  
    }  
}
```

```
class pc  
{  
    public static void main(String args[])  
    {  
        myFrame f=new myFrame();  
        f.setSize(400,200);  
        f.setVisible(true);  
    }  
}
```

Class MenuBar

This class is used to create MenuBar object.

| Constructor | Description |
|-------------|----------------------------------|
| MenuBar() | It will create a MenuBar object. |

| Methods | Description |
|--------------------|---|
| void add(Menu m) | It will add specified menu in MenuBar object. |
| void remove(int n) | It will remove menu from specified position. |
| int getMenuCount() | Return total no. of menu in MenuBar. |

Class Menu

This class is used to create pull down menu pane.

| Constructor | Description |
|--------------------|--|
| Menu(String label) | It will create Menu object with specified label. |

| Methods | Description |
|------------------------|---|
| add(MenuItem mi) | It will add MenuItem in menu. |
| int getItemCount() | Returns total no. of MenuItems in menu. |
| void remove(int index) | It remove menu at specified index. |
| addSeperator() | It is used to separate menus. |
| | |

Class MenuItem

This class is used to create MenuItems which can be added into MenuPane.

| Constructor | Description |
|------------------------|------------------------------------|
| MenuItem(String label) | Create menu item with given label. |

| Methods | Description |
|------------------------------------|--|
| void addActionListener(Listener L) | This method tells MenuItem to call a special method actionPerformed when MenuItem is selected. |
| void setLabel(String label) | It is used to set label on MenuItem. |

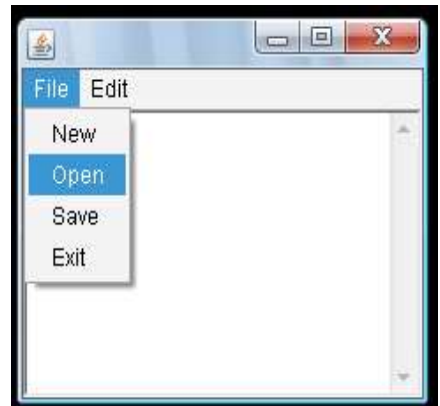
Class CheckboxMenuItem

This class is used to create MenuItem with check mark.

| Constructor | Description |
|---|---|
| CheckboxMenuItem(String label) | Create checkbox menu item with given label. |
| CheckboxMenuItem(String label, Boolean state) | Create checkbox menu item with given label and state. |

| Methods | Description |
|---|--|
| public void addItemListener(ItemListener L) | This method tells the menu item to call a special method itemStateChanged when item is selected. |
| Boolean getState() | Returns the state of the menu item. |

```
import java.io.*;      /* WAP to create simple Menu with Open Option to
import javax.swing.*;      // Open a File
import java.awt.*;
import java.awt.event.*;
class pcFrame extends Frame implements ActionListener
{MenuBar mb;
Menu mu1,mu2;
MenuItem mi;
TextArea ta;
public pcFrame()
{mb=new MenuBar();
this.setMenuBar(mb);
mu1=new Menu("File"); mb.add(mu1);
mi=new MenuItem("New");
mu1.add(mi);mi.addActionListener(this);
mi=new MenuItem("Open");
mu1.add(mi);mi.addActionListener(this);
mi=new MenuItem("Save");
mu1.add(mi);mi.addActionListener(this);
mi=new MenuItem("Exit");
mu1.add(mi);mi.addActionListener(this);
```



```

mu2=new Menu("Edit");mb.add(mu2);
mi=new MenuItem("Copy");
mu2.add(mi);mi.addActionListener(this);
mi=new MenuItem("Cut");
mu2.add(mi);mi.addActionListener(this);
mi=new MenuItem("Paste");
mu2.add(mi);mi.addActionListener(this);
ta=new TextArea();
add(ta,"Center");
}
public void actionPerformed(ActionEvent e)
{
String cmd=e.getActionCommand();
if(cmd.equals("Exit"))
    System.exit(0);
if(cmd.equals("Open"))
{
String fname=JOptionPane.showInputDialog(this,"Enter FileName");
try{
    FileInputStream fin;
    fin=new FileInputStream(fname);
    int n=fin.available();
    byte b[]=new byte[n];
    fin.read(b);
    fin.close();
    String s=new String(b);
    ta.setText(s);
    }
    catch(Exception er)
    { JOptionPane.showMessageDialog(this,er); }
}
}
}

```

Interface WindowListener

This interface is used to listen the action of windows and methods of this interface is implemented to listen window events.

| Methods | Description |
|---|--|
| public void windowOpened(WindowEvent e) | This method is called when first time window is called. |
| public void windowClosing(WindowEvent e) | This method is called when window is closed through system menu. |
| public void windowClosed(WindowEvent e) | This method is called when window is closed by call to hide. |
| public void windowIconified(WindowEvent e) | This method is called for minimized button. |
| public void windowDeIconified(WindowEvent e) | This method is called for restore button. |
| public void windowActivate(WindowEvent e) | This method is called when window is activated. |
| public void windowDeActivate(WindowEvent e) | This method is called when window is deactivated . |

MouseListener

To listen the events of mouse awt provides us two listeners .

- 1] **MouseListener**
- 2] **MouseMotionListener**

NOTE: **MouseListener** can be set for a component by calling two methods-

void addMouseListener(MouseListener L)
void addMouseMotionListener(MouseListener L)

Interface **MouseListener**

Methods of this interface are implemented to listen mouse events

| Methods | Description |
|--|--|
| public void mousePressed(MouseEvent e) | It is called when mouse button is pressed. |
| public void mouseReleased(MouseEvent e) | when pressed mouse button is released |
| public void mouseEntered(MouseEvent e) | when mouse pointer entered in component. |
| public void mouseClicked(MouseEvent e) | when mouse button is clicked. |
| public void mouseExited(MouseEvent e) | This method is called when mouse pointer is exited in container/component. |

Interface **MouseMotionListener**

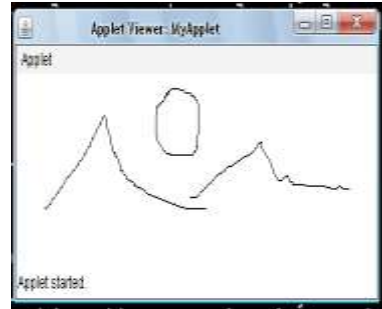
Methods of this interface is used to listen mouse motions.

| Methods | Description |
|---|--|
| public void mouseDragged(MouseEvent e) | This method is called when mouse is moved with button pressed. |
| public void mouseMoved(MouseEvent e) | This method is called when mouse is moved. |

```

import java.awt.*;          //Program to create a SketchPad Applet
import java.awt.event.*;
import java.applet.*;
public class MyApplet extends Applet
implements MouseMotionListener,
MouseListener
{
int px,py;
public void init()
{
this.addMouseListener(this);
this.addMouseMotionListener(this);
}
public void mousePressed(MouseEvent e)
{
px=e.getX();
py=e.getY();
}
public void mouseDragged(MouseEvent e)
{
Graphics g=this.getGraphics();
g.drawLine(px,py,e.getX(),e.getY());
px=e.getX();
py=e.getY();
}
public void mouseReleased(MouseEvent e) { }
public void mouseMoved(MouseEvent e) { }
public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mouseClicked(MouseEvent e) { }
}

```



class Dialog

This is the base class from which all dialog classes is derived.

```
import java.awt.*;
import java.awt.event.*;
class pcFrame extends Frame implements ActionListener
{ Button b1,b2; Label l;
public pcFrame()
{ b1= new Button("select Color");
  add(b1,"North");
  b2= new Button("exit");
  add(b2,"South");
  l= new Label("CCIT",Label.CENTER);
  add(l,"Center");
```



```
Font ft= new Font ("Courier ",Font.BOLD ,52);
l.setFont(ft);
b1.addActionListener(this);
b2.addActionListener(this);
}
public void actionPerformed(ActionEvent e)
{ String s= e.getActionCommand();
  if(s.equals("exit"))
    System.exit(0);
  if(s.equals("select Color"))
    { Colordialog cd= new Colordialog(this);
      cd.show();
      Color c= cd.getColor();
      l.setForeground(c);
    }
}
}
```

```

class Colordialog extends Dialog implements ActionListener
{Button b[]; Color scolor;
Color c[]={Color.green,Color.cyan , Color.pink ,Color.darkGray,
Color.yellow };
public Colordialog(Frame f)
{super(f,"userdefined_colordialogbox",true); // to call base class constructor
  setLayout(new GridLayout(1,c.length,0,50));
  b= new Button[c.length];
  for(int i=0;i<c.length;i++)
    {b[i]= new Button();
    b[i].setBackground(c[i]);
    add(b[i]);
    b[i].addActionListener(this);
    }
  setSize(200,200); //specifies the size of the dialog box
}
public void actionPerformed(ActionEvent e)
{Button bs=(Button)e.getSource();
scolor =bs.getBackground();
setVisible(false); // when selected color dialogbox disappear
from frame window.
}
public Color getColor()
{return scolor;
}
}

class pc // function main
{public static void main(String args[])
{pcFrame pc= new pcFrame();
pc.setSize(400,400);
pc.setVisible(true);
}
}

```

Class FileDialog

The FileDialog class displays a dialog window from which the user can select a file. Being a modal dialog , it blocks the rest of the application until the user has choosen a file.

| Constructor | Description |
|--|--|
| FileDialog(Frame parent) | Creates a file dialog for loading a file. |
| FileDialog(Frame parent,String title) | Creates a file dialog window with the specified title for loading a file. |
| FileDialog(Frame parent,String title,int mode) | Creates a file dialog window with the specified title for loading a file or saving a file. |

| Methods | Description |
|-------------------------------|---|
| getDirectory() | Gets the directory of the corresponding file dialog. |
| getFile() | Gets the file of the corresponding file dialog. |
| getMode() | Indicates whether the file dialog is in the save mode or open mode. |
| void setDirectory(String str) | Sets the directory of the file dialog window to the one specified. |
| void setFile(String file) | Sets the selected file for the corresponding file dialog window to the one specified. |
| void setMode(int mode) | Sets the mode of the file dialog. |

Class MediaTracker

The MediaTracker class is a utility class to track the status of a number of media objects. Media objects could include audio clips as well as images, though currently only images are supported.

To use a media tracker, create an instance of MediaTracker and call its addImage method for each image to be tracked. In addition, each image can be assigned a unique identifier. This identifier controls the priority order in which the images are fetched. It can also be used to identify unique subsets of the images that can be waited on independently. Images with a lower ID are loaded in preference to those with a higher ID number.

```
public Image createImage(final String imagePath)
    throws InterruptedException {
    final URL url =
GraphicsFactory.class.getResource(imagePath);
    final Image image =
Toolkit.getDefaultToolkit().createImage(url);
    // XXX Not the most efficient but sufficient for now
    final MediaTracker tracker = new
MediaTracker(component);
    tracker.addImage(image, 0);
    tracker.waitForID(0);

    return image;
}
```

| | |
|---------|---|
| void | addImage (Image image, int id) Adds an image to the list of images being tracked by this media tracker. |
| void | addImage (Image image, int id, int w, int h) Adds a scaled image to the list of images being tracked by this media tracker. |
| void | removeImage (Image image) Removes the specified image from this media tracker. |
| void | removeImage (Image image, int id) Removes the specified image from the specified tracking ID of this media tracker. |
| boolean | waitForAll (long ms) Starts loading all images tracked by this media tracker. |
| void | waitForID (int id) Starts loading all images tracked by this media tracker with the specified identifier. |
| boolean | waitForID (int id, long ms) Starts loading all images tracked by this media tracker with the specified identifier. |

Adapter classes

An **adapter class** provides the default implementation of all methods in an event listener interface. **Adapter classes** are very useful when you want to process only few of the events that are handled by a particular event listener interface.

An adapter class provides an empty implementation of all methods in an event listener interface i.e this class itself write definition for methods which are present in particular event listener interface. However these definitions does not affect program flow or meaning at all.

Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

E.g. Suppose you want to use MouseClicked Event or method from MouseListener, if you do not use adapter class then unnecessarily you have to define all other methods from MouseListener such as MouseReleased, MousePressed etc.

But If you use adapter class then you can only define MouseClicked method and don't worry about other method definition because class provides an empty implementation of all methods in an event listener interface.

For eg:

- | | |
|-------------------------------|-----------------------|
| 1. WindowListener | = > WindowAdapter |
| 2. MouseListener | => MouseAdapter |
| 3. MouseMotionListener | => MouseMotionAdapter |
| 4. KeyListener | => KeyAdapter |
| 5. FocusListener | => FocusAdapter |

Exception Handling

An exception (or exceptional event) is a problem that arises during the **execution** of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

Uncaught Exception

Before you learn how to handle exception in your program, it is useful to see what happens when you don't handle them. This simple program includes an expression that intentionally causes a divide by zero error.

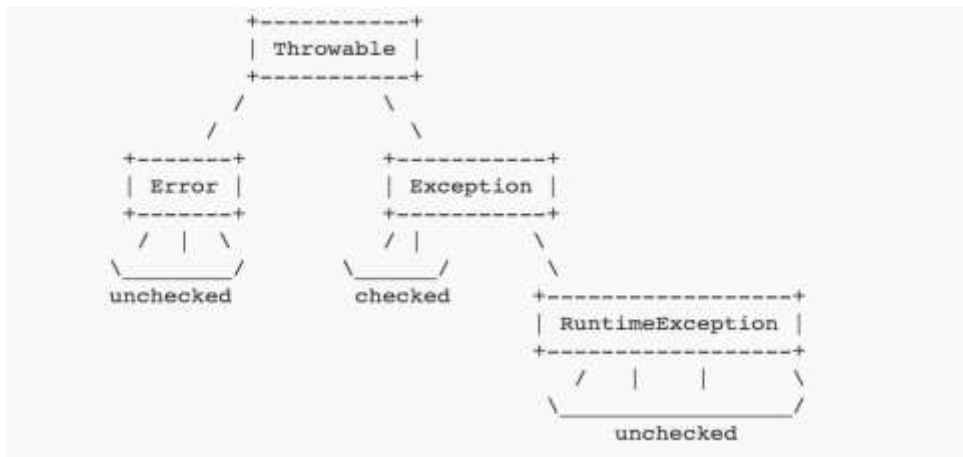
```
class ex
{
    public static void mian(String args[])
    {
        int a=4/0;
    }
}
```

When the java run time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of ex to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by default handler provided by the java run time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Checked Exception

In simple language Exception which are checked at Compile time called Checked Exception. Some these are mentioned below. If in your code if some of method throws a `checked exception`, then the method must either `handle the exception` or it must specify the exception using `throws` keyword.

1. `IOException`
2. `SQLException`
3. `DataAccessException`
4. `ClassNotFoundException`
5. `InvocationTargetException`
6. `MalformedURLException`



Unchecked Exception

Unchecked Exception in Java is those Exceptions whose handling is **NOT verified during Compile time**. These exceptions occurs because of bad programming. The program won't give a compilation error. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Simple Example: You have created online form which accepts user input. It's free text form. User may enter any wrong value in case of email field, or user name field OR phone number field. If you don't have validation at client side then there are more possibilities to get Runtime Validation Exception while running application in production. Error may throw by DB operation or converting field from one format to another.

Below are type of Unchecked Exceptions:

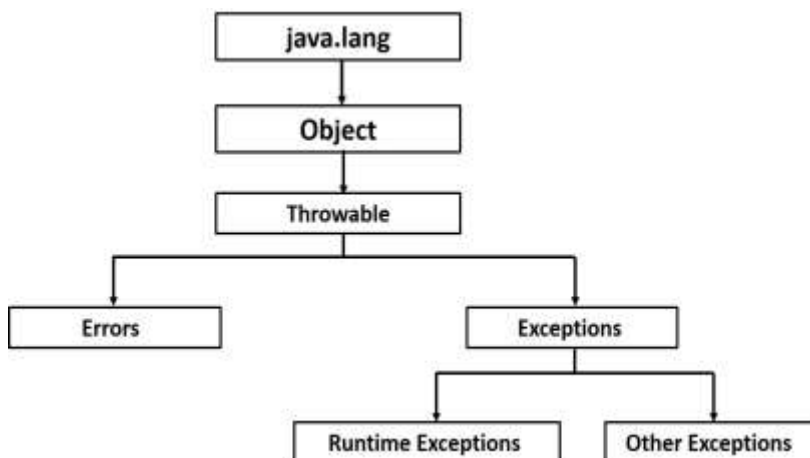
- NullPointerException
- ArrayIndexOutOfBoundsException
- IllegalArgumentException
- IllegalStateException

Exception Hierarchy

All exception classes are subtypes of the `java.lang.Exception` class. The exception class is a subclass of the `Throwable` class. Other than the exception class there is another subclass called `Error` which is derived from the `Throwable` class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The `Exception` class has two main subclasses: `IOException` class and `RuntimeException` Class.



Exceptions Methods:

Following is the list of important methods available in the Throwable class.

| SN | Methods with Description |
|----|--|
| 1 | public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | public Throwable getCause() Returns the cause of the exception as represented by a Throwable object. |
| 3 | public String toString() Returns the name of the class concatenated with the result of getMessage() |
| 4 | public void printStackTrace() Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 5 | public StackTraceElement [] getStackTrace() Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |

General form of exception handling block

```

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.
}

```

Here ExceptionType is the type of exception that has occurred.

Keyword try -

Code which we want to test for runtime errors must be in this block.

Whenever a runtime error occurs JVM creates an Exception object representing that error and program control throws into catch block.

Keyword catch -

Different catch blocks are defined after a try block to handle different types of error. An appropriate catch block is executed according to Exception object is thrown.

Keyword throw –

It is used to throw an exception object manually (i.e. ourself).

Syntax – throw ExceptionObject

It throw program control into catch block of the method.(if corresponding catch block is not found then it searches in its inner block this process continue.)

Keyword finally –

The code which we want to compulsory execute is to be placed in a finally block. The code present in this block is always executed regardless of exception thrown or not. A finally block is defined after a try block.

Keyword throws –

If a method is declared with this keyword then such a method has to be called in try block.

(when we want to handle error at compile time then this keyword is used.)

Exception Classes

JVM (java virtual machine) throws different type of exception object for different type of errors.

ArithmeticException –it is thrown when an arithmetic error occur.

NullPointerException –it is thrown when we try to access an object member without creating object.

e.g. circle c; //reference created.

c.setRadius(5); //error since object not in physical existence.

ArrayIndexOutOfBoundsException –

It is thrown when we try to access elements of an array outside its bounds.

e.g. int a[]= new int[10];

for(i=0;i<=a.length ;i++) //error there is no element at position 10.

ClassNotFoundException-

An Exception object is thrown when class is not found.

**/*exception handling program control tranfer to another block
when error occur */**

```
class err1
{
    public static void main(String args[])
    {
        System.out.println(" before call" );
        test(4,0);
        System.out.println(" after call" );
        System.out.println(" program end" );
    }

    public static void test(int a,int b)
    {
        int c;
        try    {
            System.out.println("before expression...");
            c= a/b;
            System.out.println("after expression");
        }
        catch(ArithmeticException er)
        {
            System.out.println("error message : "+er);
        }
    }
}
```

//exception handling implementing finally block.

```

class err2
{
    public static void main(String args[])
    {
        System.out.println(" before call" );
        test(4,0);
        System.out.println(" after call" );    // this is executed
        System.out.println(" program end" );    // this is executed
    }
    public static void test(int a,int b)
    {
        int c;
        try    {
            System.out.println("before expression...");
            c= a/b;
            System.out.println("after expression");
        }
        catch(ArithmeticException er)
        {
            System.out.println("error message : "+er);
        }
        finally    // this block is compulsarily executed.
        {
            System.out.println("this is compulsarily executed");
        }
    }
}

```


//program implementing ' throw ' keyword

```

class err3
{
    public static void main(String args[])
    {
        System.out.println(" before call" );
        test(4,0);
        System.out.println(" after call" );    // this is executed
        System.out.println(" program end" );    // this is executed

    public static void test(int a,int b)
    {
        try
        {
            int c;
            System.out.println("in method ...");
            // throw our Exception object.
            if(b==0) throw new ArithmeticException("exception occurred");
            c= a/b;
            System.out.println("after statement");
        }
        catch(ArithmeticException er)
        {
            System.out.println("error : "+er);
        }
    }
}

```

User-defined Exceptions:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{  
}
```

Common Exceptions

In Java, it is possible to define two categories of Exceptions and Errors.

- **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
- **Programmatic exceptions:** - These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

Write a program to accept password from user and throw 'Authentication failure' exception if password is incorrect.

```
import java.io.*;
class PasswordException extends Exception
{ PasswordException(String msg)
  { super(msg);
  }
}
class PassCheck
{ public static void main(String args[])
  {
    BufferedReader bin=new BufferedReader(new InputStreamReader(System.in));
    try
    {
      System.out.println("Enter Password : ");
      if(bin.readLine().equals("CCIT"))
      {
        System.out.println("Authenticated ");
      }
      else
      {
        throw new PasswordException("Authentication failure");
      }
    }
    catch(PasswordException e)
    { System.out.println(e);
    }
    catch(IOException e)
    { System.out.println(e);
    }
  }
}
```

Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
class Excep6{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b = 39/0;
            }
            catch(ArithmeticException e){System.out.println(e);}

            try{
                int a[] = new int[5];
                a[5] = 4;
            }
            catch(ArrayIndexOutOfBoundsException e){System.out.println(
e);}

            System.out.println("other statement);
        }
        catch(Exception e){System.out.println("handeled");}

        System.out.println("normal flow..");
    }
}
```

Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

For eg:

```
class TestExceptionPropagation1{
    void m(){
        int data=50/0;
    }
    void n(){
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        TestExceptionPropagation1 obj=new TestExceptionPropagation1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

throw vs throws

| throw | throws |
|--|---|
| Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| Throw is followed by an instance. | Throws is followed by class. |
| Throw is used within the method. | Throws is used with the method signature. |
| You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

final vs finally vs finalize

| final | finally | finalize |
|--|---|--|
| Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| Final is a keyword. | Finally is a block. | Finalize is a method. |
| <pre>class FinalExample{ public static void main(String[] args) { final int x=100; x=200;//Compile Time Error }}</pre> | <pre>class FinallyExample { public static void main(String[] args) { try{ int x=300; } catch(Exception e) {System.out.println(e);} Finally {System.out.println("finally bk");} }}</pre> | <pre>class FinalizeExample { public void finalize() {System.out.println("finalize");} public static void main(String[] args) { FinalizeExample f1=new FinalizeExample(); FinalizeExample f2=new FinalizeExample(); f1=null; f2=null; System.gc(); }}</pre> |

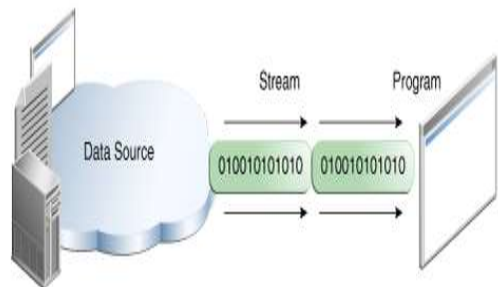
I/O Streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

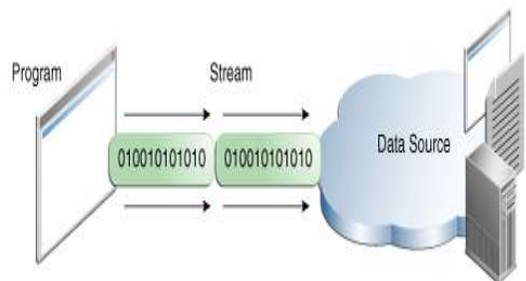
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

A stream is a sequence of data.

InputStream: A program uses an *input stream* to read data from a source, one item at a time:



OutputStream: A program uses an *output stream* to write data to a destination, one item at a time:



The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

Class File -

An object of this type represents a file or directory.

Constructors -

File(String path) -

File(String directorypath,String filename) -

File(String directorypath) -

File(File dir,String filename) -

Methods-

long length() - returns length or size of file in bytes.

e.g. long n= f4.length();

boolean exists()- returns true if path exists i.e. File or Directory .

e.g. if(f2.exists())

boolean isFile() - returns true if object represents a File.

boolean isDirectory() - returns true if object represents a Directory.

boolean mkdir()- will create the directory

e.g. File f= new File("c:/jdk/bin/mydir");

f.mkdir(); // will create directory in specified

boolean mkdirs() - will create all the required directory.

String[] list() It returns list of all the file names if object represents a directory.

class InputStream

An object of this type represents an InputStream .It is an abstract class . Different InputStream classes are derived from this class.It provides us methods to read data from an InputStream.

Methods -

int read() - it reads a single byte from an InputStream and return int value.

int read(byte b[]) - it read b.length bytes from InputStream and store them into byte array.

int read(byte b[],int offset,int n) - it reads n byte from the stream and stores it into byte array

Note - All the above methods return -1 at the end of stream.

int available() - it return no. of bytes that are available for reading .

void close() - it closes InputStream .

class FileInputStream

This class is derived from class InputStreeam .It provides features to read data from an FileStream.

Constructors -

FileInputStream(String filename) -

FileInputStream (File f) -

//WAP read adata from file byte by byte.

```
import java.io.*;
class file1
{public static void main(String args[]) throws Exception
{
    int ch;
    FileInputStream fin;
    fin = new FileInputStream ("e://jdk1.3/bin/xscroll.java");
    while((ch=fin.read())!=-1)    //read a single byte
        { System.out.print((char) ch);    // print char which is read as byte.
        }
    fin.close();    //close the file.
}
}
```

//WAP read a whole data from file and print.

```
import java.io.*;
class file2
{
    public static void main(String args[])throws Exception
    {
        FileInputStream fin;
        fin = new FileInputStream ("e://jdk1.3/bin/xscroll.java");
        int n=fin.available();    // get file size.
        byte b[] = new byte[n];    //create a buffer of file size
        fin.read(b);    //read a buffer.

        fin.close();    //close the file.

        String s= new String (b);    //convert byte to String.
        System.out.print(s);    // print char which is read as byte.
    }
}
```

Class OutputStream

It is an abstract class . Different OutputStream classes are derived from this class.It provides us methods to write data into a stream.

Methods -

void write(int b) - it writes a byte(i.e. a single byte) into a OutputStream .

void write(byte b[]) - it will write all the bytes from byte[] into OutputStream.

void write (byte b[],int offset,int n) - it writes n bytes starting from specified offset from byte[] into OutputStream.

void close() - it closes OutputStream .

class FileOutputStream

This class is derived from class OutputStream .It provides features to write data into FileStream.We can use all the methods OutputStream class.

Constructors -

FileOutputStream(String filename) -

FileOutputStream (File f) -

Class Reader

Abstract class for reading character streams. The only methods that a subclass must implement are `read(char[], int, int)` and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Methods

| | |
|------------------|--|
| abstract void | <code>close()</code> Closes the stream and releases any system resources associated with it. |
| int | <code>read()</code> Reads a single character. |
| int | <code>read(char[] cbuf)</code> Reads characters into an array. |
| abstract int | <code>read(char[] cbuf, int off, int len)</code> Reads characters into a portion of an array. |
| int | <code>read(CharBuffer target)</code> Attempts to read characters into the specified character buffer. |
| boolean | <code>ready()</code> Tells whether this stream is ready to be read. |
| void | <code>reset()</code> Resets the stream. |
| long | <code>skip(long n)</code> Skips characters. |

Class Writer

Abstract class for writing to character streams. The only methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Methods

| | | |
|---------------|--|---|
| Writer | append (char c) | Appends the specified character to this writer. |
| Writer | append (CharSequence csq) | Appends the specified character sequence to this writer. |
| Writer | append (CharSequence csq, int start, int end) | Appends a subsequence of the specified character sequence to this writer. |
| void | close () | Closes the stream, flushing it first. |
| void | flush () | Flushes the stream. |
| void | write (char[] cbuf) | Writes an array of characters. |
| void | write (char[] cbuf, int off, int len) | Writes a portion of an array of characters. |
| void | write (int c) | Writes a single character. |
| void | write (String str) | Writes a string. |
| void | write (String str, int off, int len) | Writes a portion of a string. |

Class FileReader

This class inherits from the Reader class. FileReader is used for reading streams of characters.

Constructors

FileReader(File file)

This constructor creates a new FileReader, given the File to read from.

FileReader(String fileName)

This constructor creates a new FileReader, given the name of the file to read from.

Class FileWriter

This class inherits from the Writer class. The class is used for writing streams of characters.

Constructors

FileWriter(String fileName)

This constructor creates a FileWriter object, given a file name.

FileWriter(String fileName, boolean append)

This constructor creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

```

import java.io.*;

public class FileRead{

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

        File file = new File("Hello1.txt");
        // creates the file
        file.createNewFile();
        // creates a FileWriter Object
        FileWriter writer = new FileWriter(file);
        // Writes the content to the file
        writer.write("This\n is\n an\n example\n");
        writer.flush();
        writer.close();

        //Creates a FileReader Object
        FileReader fr = new FileReader(file);
        char [] a = new char[50];
        fr.read(a); // reads the content to the array
        for(char c : a)
            System.out.print(c); //prints the characters one by one
        fr.close();
    }
}

```


class PrintStream

A PrintStream provides a wrapper for an OutputStream. This class provides us simple methods to write different types of values in a OutputStream object.

Constructor –

PrintStream (OutputStream out) –

It creates a PrintStream object by wrapping a OutputStream object in it.

Methods –

void print(String s) – Will print the String.

void print(int n) – Will print the integer value.

void println(String s) – Will print the String with line feed.

void println(int n) – Will print the integer value with line feed.

Similar methods for different datatypes.....

import java.io.*; //program to print int values int file mynos.txt

class Test

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

{int a[]={5454,4534,1234,2345};

FileOutputStream fout= new FileOutputStream("c:/mynos.txt");

PrintStream ps= new PrintStream(fout);

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

ps.println(a[i]);

ps.close();

fout.close();

}

}

Class DataOutputStream

It provides us an wrapper to write different types of primitive values into an OutputStream.

Constructor –

DataOutputStream(OutputStream out) –

Will create a DataOutputStream object by wrapping OutputStream object in it.

Methods –

void writeBytes(String s) – will write a string into OutputStream.

void writeInt(int n) –will write int value as 4 bytes into OutputStream .

void writeLong(long n) –will write long value as 8 bytes into OutputStream .

similar methods for different datatypes.....

```
import java.io.*;    // program to store int values in binary format into file
class Test
{public static void main(String args[])
    {int a[]={5454,4534,1234,2345};
      FileOutputStream fout= new FileOutputStream("c:/x1.txt");
      DataOutputStream ds= new DataOutputStream (fout);
      for(int i=0;i<a.length;i++)
          ds.writeInt(a[i]);
      ds.close();
      fout.close();
    }
}
```

Class DataInputStream

It provides us an wrapper to read different types of primitive values into an InputStream.

Constructor –

DataStream(InputStream in) –

Will create a DataInputStream object by wrapping InputStream object in it.

Methods –

String readline() – will read a Line from InputStream.

int readInt() –will read 4 bytes from InputStream & will return it as int.

int readLong() –will read 8 bytes from InputStream & will return it as long.

similar methods for different datatypes.....

```
import java.io.*;
class Test
{public static void main(String args[])
    {try{
        File f= new File("c:/x1.txt");
        int count = (int)f.length()/4;           //total no. of elements.
        FileInputStream fin= new FileInputStream(f);
        DataInputStream din= new DataInputStream (fin);
        for(int i=0;i<count;i++)
            {int n=din.readInt(); //method of DataInputStream
              System.out.println(n);
            }
        din.close();
        fin.close();
    }
    catch(Exception er)
        { System.out.println("Error : "+er);
        }
    }
}
```

Object Serialization

Serialization in java **is a mechanism of writing the state of an object into a byte stream.**

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

```
import java.io.Serializable;
public class Student implements Serializable{
    int id;
    String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

```
import java.io.*;
class Persist{
    public static void main(String args[])throws Exception{
        Student s1 =new Student(211,"ravi");
```

```
        FileOutputStream fout=new FileOutputStream("f.txt");
        ObjectOutputStream out=new ObjectOutputStream(fout);
```

```
        out.writeObject(s1);
        out.flush();
        System.out.println("success");
    }
}
```

Deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization.

ObjectInputStream class

An `ObjectInputStream` deserializes objects and primitive data written using an `ObjectOutputStream`.

For eg:

```
import java.io.*;
class Depersist{
    public static void main(String args[])throws Exception{

        ObjectInputStream in=new ObjectInputStream(new FileInputStream(
            "f.txt"));
        Student s=(Student)in.readObject();
        System.out.println(s.id+ " "+s.name);

        in.close();
    }
}
```

Multi Threading

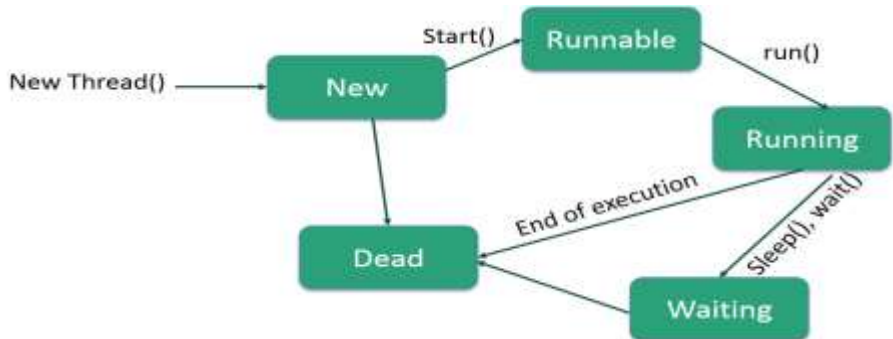
Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

A Thread is a path of Execution . A multithreaded application contains multiple execution paths in a program . Due to this multiple parts of program are concurrently executed. Java makes it easy to define and work with multiple threads of execution within a program. `java.lang.Thread` is the fundamental thread class in the Java API. There are two ways to define a thread. One is to subclass `Thread`, override the `run()` method, and then instantiate your `Thread` subclass. The other is to define a class that implements the `Runnable` method (i.e., define a `run()` method) and then pass an instance of this `Runnable` object to the `Thread()` constructor. In either case, the result is a `Thread` object, where the `run()` method is the body of the thread. When you call the `start()` method of the `Thread` object, the interpreter creates a new thread to execute the `run()` method. This new thread continues to run until the `run()` method exits, at which point it ceases to exist. Meanwhile, the original thread continues running itself, starting with the statement following the `start()` method.

Life Cycle of a Thread

A thread goes through various stages in its life cycle.



- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead):** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Class Thread

Constructors

Thread() : Allocates a new Thread object.

Thread(Runnable) : Allocates a new Thread object.

Thread(Runnable, String) : Allocates a new Thread object.

Thread(String) : Allocates a new Thread object.

Methods

activeCount() : Returns the current number of active threads in this thread group.

getPriority() : Returns this thread's priority.

isAlive() : Tests if this thread is alive.

isDaemon() : Tests if this thread is a daemon thread.

join() : Waits for this thread to die.

join(long) : Waits at most millis milliseconds for this thread to die.

join(long, int) : Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

resume() : Resumes a suspended thread.

run() : If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.

setDaemon(boolean) : Marks this thread as either a daemon thread or a user thread.

setName(String) : Changes the name of this thread to be equal to the argument name.

setPriority(int) : Changes the priority of this thread.

sleep(long) : Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

start() : Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

stop() : Forces the thread to stop executing.

stop(Throwable) : Forces the thread to stop executing.

suspend() : Suspends this thread.

yield() : Causes the currently executing thread object to temporarily pause and allow other threads to execute.

For eg: To create a thread by extends class Thread

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
}
```

```
class demo{
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
}
}
```

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

To set/get Thread Priority we can use Methods

void setPriority(int) : Changes the priority of this thread.
int getPriority() : Returns this thread's priority.

For eg:Creating thread by implementing Runnable interface

1. Let your class implement "Runnable" interface.
2. Now override the "public void run()" method and write your logic there (This is the method which will be executed when this thread is started).

```
public class MyRunnableTask implements Runnable {  
    public void run() {  
        // do stuff here  
    }  
}
```

That's it, now you can start this thread as given below

1. Create an object of the above class
2. Allocate a thread object for our thread
3. Call the method "start" on the allocated thread object.

```
class demo{  
    public static void main(String args[]){  
        MyRunnableTask t1=new MyRunnableTask ();  
        t1.start();  
    }  
}
```

Thread Synchronization

When using multiple threads, you must be very careful if you allow more than one thread to access the same data structure. Consider a Non Shareable object such as printer is being used by multiple threads to print. Preventing this problem is called *thread synchronization*. The basic technique for preventing two threads from accessing the same object at the same time is to require a thread to obtain a lock on the object before the thread can modify it. While any one thread holds the lock, another thread that requests the lock has to wait until the first thread is done and releases the lock. Every Java object has the fundamental ability to provide such a locking capability. The easiest way to keep objects thread-safe is to declare all sensitive methods synchronized.

Thread Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock. Deadlock occurs when multiple threads need the same locks but obtain them in different order.


Dead Lock Prevention

- Lock ordering is a simple yet effective deadlock prevention mechanism.
- Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry.

Difference between threads and processes


| Process | Thread |
|--|--|
| A process has separate virtual address space. Two processes running on the same system at the same time do not overlap each other. | Threads are entities within a process. All threads of a process share its virtual address space and system resources but they have their own stack created. |
| Every process has its own data segment | All threads created by the process share the same data segment. |
| Processes use inter process communication techniques to interact with other processes. | Threads do not need inter process communication techniques because they are not altogether separate address spaces. They share the same address space; therefore, they can directly communicate with other threads of the process. |
| Process has no synchronization overhead in the same way a thread has. | Threads of a process share the same address space; therefore synchronizing the access to the shared data within the process's address space becomes very important. |
| Child process creation within from a parent process requires duplication of the resources of parent process | Threads can be created quite easily and no duplication of resources is required. |

Diploma in Data Science



- PHP + My-SQL
- Python + Django
- Data Analysis
- Artificial Intelligence
- Machine learning
- Apache Hadoop

- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON



With 3 Major Projects



The advertisement features a dark blue header with the title 'Diploma in Java' in white. To the right is the Java logo. Below the header, a light blue box contains a list of topics. To the right of this box is a red banner with the word 'FREE' in white. At the bottom, a dark blue bar contains the text 'With 3 Major Projects' in white.

Diploma in Java

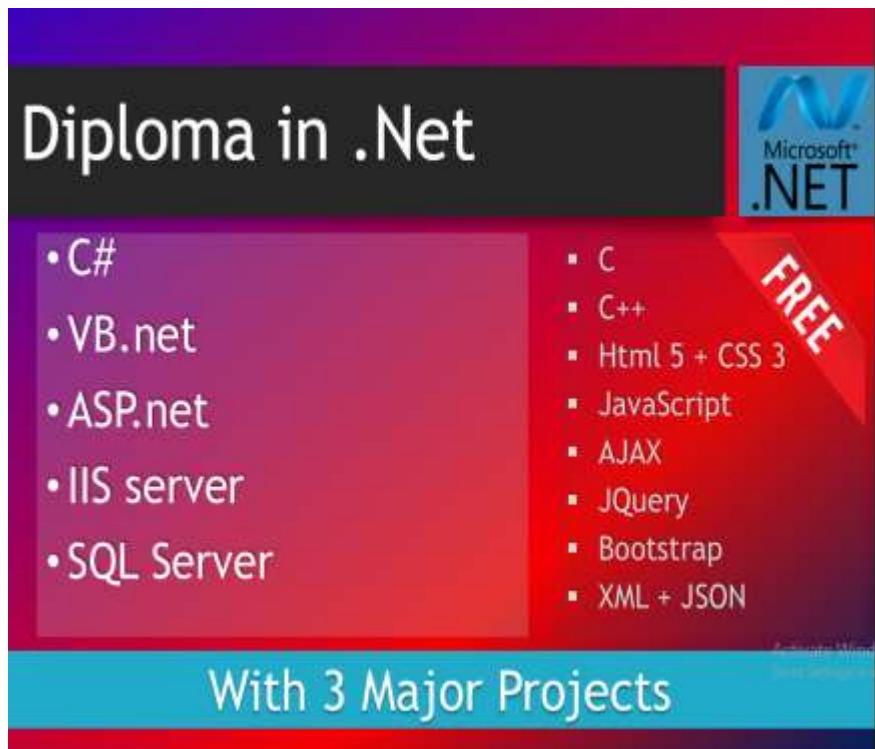


- Core Java
- Advance Java [J2EE]
- Android Programming
- Hibernate
- Spring
- My-SQL

- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON


FREE

With 3 Major Projects



The advertisement features a dark blue header with the title 'Diploma in .Net' in white. To the right is the Microsoft .NET logo. Below the header, a red diagonal banner with the word 'FREE' in white is positioned over a list of technologies. The technologies are divided into two columns: the left column lists C#, VB.net, ASP.net, IIS server, and SQL Server; the right column lists C, C++, Html 5 + CSS 3, JavaScript, AJAX, JQuery, Bootstrap, and XML + JSON. At the bottom, a blue banner contains the text 'With 3 Major Projects'.

Diploma in .Net



FREE

- C#
- VB.net
- ASP.net
- IIS server
- SQL Server
- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON

With 3 Major Projects



Diploma in Web Technology

React logo

- PHP + MySQL
- React
- Node.js
- Amazon Web Services
- Kotlin [Android]

- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON

FREE

With 3 Major Projects

© 2019 by TAFE NSW