## B.SC.,/B.C.A. FOURTH SEMESTER

## JAVA PROGRAMMING

**UNIT I:** Introduction to Java - Features of Java - Object Oriented Concepts - Lexical Issues - Data Types - Variables - Arrays - Operators - Control Statements.

**UNIT II:** Classes - Objects - Constructors - Overloading method - Access Control- Static and Final methods - Inner Classes - String Objects - String Class -String Buffer - Char Array - Java Utilities - Inheritance - Overriding methods - Using super-Abstract class.

**UNIT III:** Packages - Access Protection - Importing Packages - interfaces - Exception Handling - Throw and Throws - Thread - Synchronization - Messaging - Runnable Interface - Inter thread Communication - Deadlock - Suspending, Resuming and stopping threads - Multithreading.

**UNIT IV:** I/O Streams: Stream classes – Byte stream classes - Character stream classes - File Streams – Using File class – I/O exceptions – Creation of file – Reading or writing characters/bytes – Random access files.

**UNIT V:** Applets – Preparing to write applets – Building Applet code – Applet life cycle – Applet tag – Passing parameters to Applets - Working with windows using AWT Classes - AWT Controls - Layout Managers and Menus.

**TEXT BOOKS:**


1. **E.Balagurusamy**, "*Programming with Java*", Fourth Edition, 2010, Tata McGraw-Hill.
2. **P Radha Krishna**, "*Object Oriented Programming through Java*", Second Edition, 2007, Universities Press.


**REFERENCE BOOKS:**


1. **K. Arnold and J. Gosling**, "*The Java Programming Language*", Second Edition, 1996, Addison Wesley.
2. **P. Naughton and H. Schildt**, "*Java2 (The Complete Reference)*", Eight Edition, 2005, Tata McGraw-Hill.
3. **Kathy Sierra and Bert Bates**, "*Head First Java*", Second Edition, 2003, Oreilly.


**E- REFERENCES:**

- www.tutorialspoint.com/java/java-quick-guide.htm
- www.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html
- www.tutorialspoint.com/java/java_overview.htm

## UNIT 1

**Introduction to Java - Features of Java - Object Oriented Concepts - Lexical Issues - Data Types - Variables - Arrays - Operators - Control Statements.**

### INTRODUCTION TO JAVA

Java programming language was originally developed by Sun Microsystems, which was initiated by **James Gosling in 1991**. It was initially called "Oak" after an oak tree that stood outside Gosling's office, and was renamed as "Java" in 1995. Java includes the best aspects of earlier programming languages such as C and C++. But it also leaves out those things that we all hated about C and C++ like multiple inheritance, operator overloading and pointers. It has built-in multimedia capabilities and it allows creating powerful applications and made the multimedia presentations easier than ever.

A Java compiler converts the Java source code into a binary program consisting of byte codes. Byte codes are optimized set of instructions designed to be executed by the Java run-time system, which is called the **Java Virtual Machine (JVM).** JVM is an interpreter for byte code. This interpreter inspects and deciphers the byte codes and executes the actions that the byte codes specify. A Java interpreter can run stand-alone or it can be a part of a web browser such as Netscape Navigator or IE where it can be invoked automatically to run applets in a web page.

Translating a Java program into byte code makes it easier to run a program in a wide variety of environments. Only JVM needs to be implemented for each platform. Although the details of JVM differ from platform to platform, all interpret the same Java byte code. When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However with Java, the difference is not great. The use of byte code enables the Java run-time system to execute program much faster than you might expect. Sun has just completed its **Just In Time (JIT)** compiler for byte code, which is included in Java 2. JIT is a part of JVM and it compiles the byte code into executable code in real time.

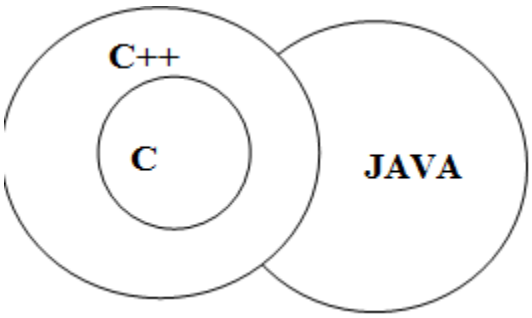
**Differences between Java and C:**

| C | Java |
|---|---|
| Procedural language | Object-Oriented language |
| Compiled | Interpreted |
| Top-Down approach | Bottom-up approach |
| Pointers | No Pointers |
| No overloading | Has overloading |
| Preprocessors | No preprocessors |
| Errors and crashes occur | Explicit exception handling |

**Differences between Java and C++:**

1. No operator overloading
2. No template classes
3. No direct multiple inheritance ( but a new concept called "interface")
4. No support to global variables
5. No pointers

6. No destructor function (but a new function called “finalize”)
7. No header files




## SALIENT FEATURES OF JAVA

The inventors of Java wanted to design a language which could offer solutions to some of the problems encountered in modern programming. They wanted the language not only reliable, portable and distributed but also simple, compact and interactive. Some of the Java features to state are:

1. **Compiled and Interpreted:** Normally a language is compiled or interpreted. But Java combines these approaches making Java a two-stage system. First Java compiler translates source code into bytecode instructions, which are not machines instructions. In second pass, java Interpreter generates machines code that can be directly executed by the machines that is running the Java Program. So Java is both a compiled and an interpreted language. Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
2. **Platform independent and Portable:** Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run. Java is guaranteed to be Write Once, Run Anywhere. Changes and upgrades in the operating systems, processors and system resources will not force any changes in Java Programs.
3. **Object Oriented:** Java is a pure object oriented language since in Java everything is an Object. All program code and data reside within objects and classes. Java is simple and easily extended since it is based on the Object model.
4. **Robust and Secure:** Since Java ensures reliable code, strict compile time and runtime checking of data types, exception handling, garbage-collected language etc, it is robust in nature. With Java's secure feature it enables to develop virus-free, tamper-free systems. The absence of pointers in Java and no virus can communicate applet ensures that programs cannot gain access to memory locations without proper authorizations.
5. **Distributed:** Java is designed for the development of applications on networks since it is a distributed language. This enables multiple programmers at multiple remote locations to collaborate and work on a same project.

6. **Simple, small and familiar:** Java is designed to be easy to learn. If we understand the basic concept of OOP Java would be easy to master since it uses most of the constructs from C and C++. Java is a simplified version of C++.
7. **Multi-threaded:** With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications with synchronization.
8. **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time. Java is capable of dynamically linking in new class libraries, methods and objects. Java programs support functions of languages of C and C++ called as native methods.
9. **High Performance:** With the use of Just-In-Time compilers Java enables high performance. Moreover J2SE 5.0 assures a significant increase in scalability and performance by improving the startup time and reducing the amount of memory used in Java 2 runtime environment.
10. **Ease of development:** Various features in Java reduce the work of the programmer by shifting the responsibility of creating the reusable code to the compiler and as a result the source code is free from bugs because the errors made by the compiler are less when compared to those made by programmers.

## OBJECT ORIENTED CONCEPTS

Java is a pure Object Oriented Language. As a language that has the Object Oriented features, Java supports the fundamental concepts like polymorphism, inheritance, encapsulation, abstraction, classes, methods, message passing etc.

## JAVA ENVIRONMENT

Java environment consists of number of development tools, classes and methods. The development tools are the part of system called as Java Development Kit (JDK) and the classes and methods are part of Java Standard Library (JSL) or Application Programming Interface (API).

**JDK**: Collection of tools that are used for developing and running Java programs. They are:

- appletviewer (for viewing applets)
- javac (java compiler converts source code to bytecode)
- java (java interpreter runs the application)
- javap (java disassembler arranges the bytecode file to program)
- javah (for c header files to implement native methods)
- javadoc (HTML documents)
- jdb (java debugger to find errors)

**API:** Hundred of classes are made into packages. Some of them are:

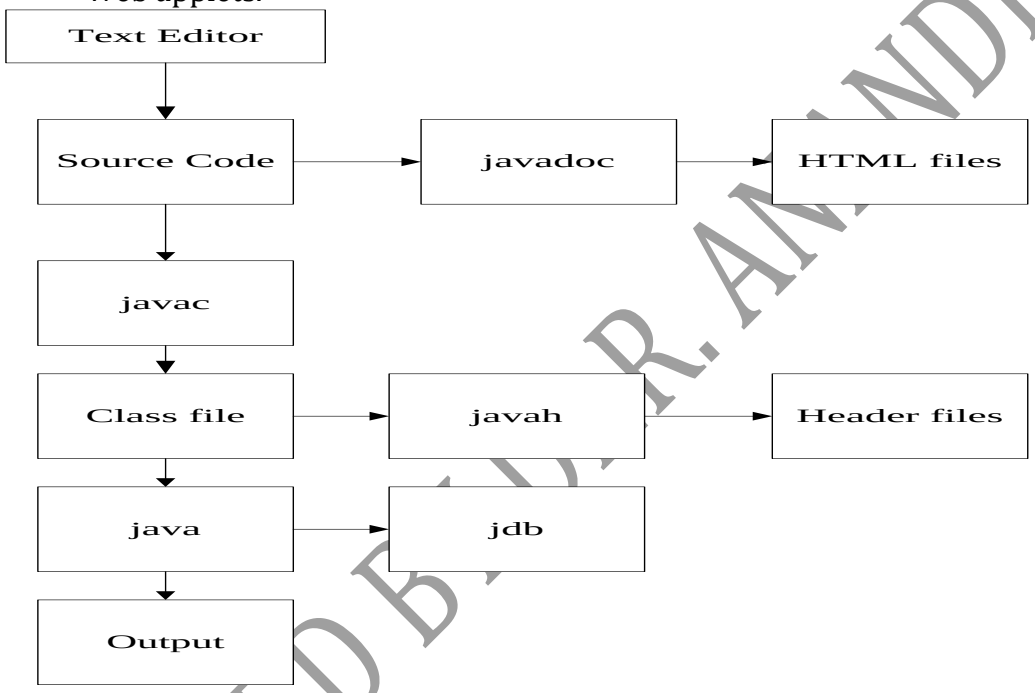
- Language support package
- Utilities package
- Input/Output package
- Networking package
- AWT package
- Applet package

**JRE (Jave Runtime Environment)** facilitates the program execution. It consists of

- **JVM (Java Virtual Machine)** which interprets Java bytecode and generates desired output. Bytecode and JVM are highly portable. It is a machine that does not exist. It actually exists in computer memory i.e. it is a computer within a computer.
- **Runtime Class Libraries** is required during the program execution.
- **User interface toolkits** like AWT, Swing etc.
- **Deployment technologies** like Java Plug-in, Java Web Start etc.

We can develop two types of Java programs as

- Stand-alone applications
- Web applets.




## GENERAL FORMAT OF A JAVA PROGRAM

Java is a free-form language but the users should not take the advantage of it. The general format of a Java Program is

1. **Documentation Section:** // or /*....*/ are the symbols used to denote the comment lines. This section will generally speak about the details of the program.
2. **Package Statement** declares the package name.
3. **Import statement** will import the set of classes to be used in the program.
4. **Interface statement** will be used if the program uses the concept of multiple inheritances.
5. **Class definitions** will enclose the primary and essential elements of Java program. A program may have multiple class definitions.
6. **Main method class**

```
{
        Main method definition
}
```

Every Java program should have a main method. Execution starts here. Only one main method should be used in a program.

## FIRST JAVA PROGRAM

Let us look at a simple code that would print the words *Hello World*.

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

Let us look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open notepad and type the code as above.
2. Save the file as: First.java.
3. Open a command prompt window and go to the directory where we saved the class. Assume its C:\.
4. Type ' javac First.java ' and press enter to compile code. If there are no errors in code the command prompt will take we to the next line.
5. Now type ' java First ' to run program.
6. We will be able to see ' Hello World ' printed on the window.

```
C : > javac First.java
C : > java First
Hello World
```

**Basic Syntax:**

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity -** Java is case sensitive which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names -** For all class names the first letter should be in Upper Case. If several words are used to form a name of the class each inner words first letter should be in UpperCase. Example *class MyFirstJavaClass*
- **Method Names -** All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case. Example *public void myMethodName()*
- **Program File Name -** Name of the program file should exactly match the class name holding the main method. When saving the file we should save it using the class name (Remember java is case sensitive) and append '.java' to the end of the name. (if the file name and the class name do not match, program will not compile). Example: Assume 'First' is the class name. Then the file should be saved as 'First.java'.
- **public static void main(String args[]) -**Java program processing starts from the main() method which is a mandatory part of every java program. **public** means unprotected access specifiers. **static** means this main method should be called before any object is created i.e. main method attached to class and not to the object. **void** means the main method will not return anything. **main** is the keyword of the function.

## DATA TYPES IN JAVA

One of the fundamental concepts of any programming language is data types. Data types define the storage methods available for representing information along with how the information is interpreted. Variables are nothing but reserved memory locations to store values. This means that when we create a variable we reserve some space in memory. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, we can store integers, decimals, or characters in these variables. There are two data types available in Java:

1. **Primitive Data Types or Simple Data Types** are core types not derived from any other types.
2. **Composite Data Types** are based on Simple ones.

## PRIMITIVE DATA TYPES

There are **eight** primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

1. **Integer** represents only the whole numbers. It includes byte, short, int, long.
    1. **int** represents integer values (4 bytes).
    2. **byte** is the smallest integer type useful for file handling and stream handling (1 byte)
    3. **short** represents whole numbers of smaller length (2 bytes).
    4. **long** represents values of higher range (8 bytes).
    5. Make integers as long by appending L or l at end.
2. **Floating-point numbers** includes float and double used to represent the fractional numbers.
    1. **float** represents commonly used floating point numbers.
    2. **double** represents floating numbers of larger order used in mathematical and scientific calculations for larger accuracy.
    3. Math functions return double value.
    4. Floating point supports NAN (Not A Number).
    5. Append F or f to make a number float explicitely.
3. **Boolean** represents true or false which are useful during comparison. These values are not changed from this type to other. The variables are declared by the keyword boolean.
4. **Character** includes char used to represent a character or a symbol. Character must be a single character of ASCII or UNICODE, which is a superset of ASCII and solves multi-language problems. ASCII is 8-bit length with 256 characters. For regional languages, since 256 characters are not enough, we go for UNICODE in which first 256 characters are ASCII and the remaining dealt with the individual language.

| **Type** | Boolean | Byte | Char | Short | Int | Float | Long | Double |
|---|---|---|---|---|---|---|---|---|
| **Size in bits** | 1 | 8 | 16 | 16 | 32 | 32 | 64 | 64 |

**Eg:**

```
class Chardemo
{
  public static void main(String args[])
     {
     char ch1, ch2;
      ch1 = 88;
      ch2 = "Y";
      System.out.println("ch1 and ch2 : " + ch1+" "+ch2);
    }
}
```

The output is

ch1 and ch2 : X Y

where 88 is the ASCII value that corresponds to X.

## VARIABLES IN JAVA

Every variable should be declared before they are used. Declaration of variables does three things as:

a. It tells the compiler what the variable name is
b. It specifies what type of data the variable will hold
c. The place of declaration reveals the scope of the variable

A variable can store can data type value and it should follow the rules of Java for naming of identifiers. Variables are separated by commas and declaration statement must end with semicolon.

**Syntax:** datatype variable1, variable2,……variablen;

Eg: int x,y; char e;

During declaration variables, we can assign the values to them.

Eg: int a = 8, b = 7; double x = 49.90; char c = 'j';

**Dynamic initialization:** The variables can be initialized dynamically, using any expression valid at the time the variable is declared.

```
Class Leng
{
  public static void main(String args[ ])
{
      double a = 3.0 , b = 4.0;
      double c = Math.sqrt(a * a  +  b * b) ;
      System.out.println(" Hypotenuse " +c);
 }
}
```

a and b are declared as constants, c is initialized dynamically to the length of the hypotenuse.

**Printing values of variables:** A computer program is written to manipulate a set of values and display the results on the screen. Two methods used to display the values of the variables are print() and println(). The method print() prints the output on one line until a newline character is encountered whereas the method println() takes the information provided and inserts one carriage return in every line.

**Eg:**
System.out.print("Hello");
System.out.print("Java"); will print the results as Hello Java in one line.
**Eg;**
System.out.println("Hello");
System.out.println("Java"); will print the results as
Hello
Java
in two lines
There are three kinds of variables in Java:

1. Local variables
2. Instance variables or member variables
3. Class/static variables

**Type casting:** When we assign an int value to a byte variable the conversion will not be performed automatically as a byte is smaller than an int. The conversion is called a **narrowing conversion** as we are explicitly making the value narrower to fit into the target type. A **cast** is a explicit type conversion. The form is

**(target-type)value**

The target-type specifies the desired type to convert the specified value to.
Eg:

```
int  a;
byte b;
b = (byte) a;
```

**Type promotion rules:**

1. All bytes and short values are promoted to int.
2. If one operand is long, the whole expression is promoted to long.
3. If one operand is float, the entire expression is promoted to float.
4. If any of the operands is double, the result is double.

## LEXICAL ISSUES OR JAVA TOKENS

There are many atomic elements of Java statements. The smallest individual unit in a program is called as Token. Java programs are a collection of whitespace, identifiers, comments, literals, operators, separators and keywords.

1. **Whitespaces:** Java is a free-form language (ie) we do not need to follow any special indentation result. In Java, whitespace is a space, tab or newline.
2. **Identifiers:** Identifiers are user-defined names used for class names, method names or variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar sign characters. Eg: AvgTemp, count, $calculate, s5, value, display. An identifier must not begin with a number because it leads to invalid identifier. Eg: 8y, yes/no.

   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

3. **Literals:** A constant value in Java is created by using a literal representation. A literal is allowed to use anywhere in the program. A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable. Integer literals, floating-point literals, character literals, string literals, Boolean literals are the various types of literals. For example:

```
byte a = 68;
char a = 'A'
```

Byte, int, long, and short can be expressed in decimal (base 10), hexadecimal(base 16) or octal(base 8) number systems as well. Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals. Floating point literals must be suffied by F, f, D or d. If suffixes are omitted in a floating point number, it is treated as double. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
5727.979F
452020.208d
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters.
For example:

```
char a = '\u0001';
String a = "\u0001";
```

Boolean literals are obviously going to take the value true or false. Java language supports few special escape sequences for String and char literals as well. They are:

| Notation | Character represented |
|---|---|
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |

| | |
|---|---|
| \s | Space (0x20) |
| \t | tab |
| \" | Double quote |
| \' | Single quote |
| \\ | backslash |
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

4. **Comments:** The contents of a comment are ignored by the compiler. A comment describes or explains the operation of the program to anyone who is reading its source code. The comment describes the program. In java there are three types of comments. They are single-line, multi-line and documentation comment.
    - **Single-line comment**: Two slashes characters are the single-line comment i.e. //. This type of comment is called a "slash-slash" comment.
      // This comment extends to the end of the line.
    - **Multi-line comment**: To add a comment of more than one line, we can precede our comment using /* and end with delimiter */.
      /*.....the multi-line comments
      are given with in this comment
      .....
      .....*/
5. **Keywords:** In Java, a keyword has a predefined meaning in the language. Because of this, programmer cannot use keywords as names for variables, methods, classes, or as any other identifier. They always appear in lower case. Eg: boolean, continue, else, default etc.
6. **Operators:** Operators perform operations on the operands. Some operators need one (unary operator) while others require two operands (binary operators). Operators can be logical operators, relational operators and arithmetic operators. Eg: +, -, /, <.> etc.
7. **Separators:** Separators are used to terminate statements. In Java there are few characters are used as separators.

| Symbol | Purpose |
|---|---|
| **( )** | Used to specify the list of parameters in method definition and invocation |
| **{}** | To represent the values of initialized arrays. To define a block code for classes, methods and local scopes. |
| **[ ]** | Use to declare array types. . Also used when dereferencing array values. |
| **;** | Used to terminate statements |
| **,** | Separates consecutive identifiers and also used to chain statements together inside a for statement |
| **.** | Used to separate package names from sub packages and Classes and used to separate a variable or method from a reference variable. |

# JAVA ARRAYS

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, ..., and number99, we declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

1. **Declaring Array Variables:** We must declare a variable to refer the array to use in a program. Here is the syntax for declaring an array variable:

```
datatype[] arrayname;   or datatype arrayname[];
```

**Note:** The style **datatype[] arrayname** is preferred. The style **datatype arrayname[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

**Example:** The following code snippets are examples of this syntax

```
double[] myList;  or double myList[];
```

2. **Creating Arrays:** We can create an array by using the new operator with the following syntax:

```
arrayname = new datatype[arraySize];
```

The above statement does two things:

- It creates an array using new datatype[arraySize];
- It assigns the reference of the newly created array to the variable arrayname.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below: Operator "new" is used to allocate space for an array.

```
datatype[] arrayname = new datatype[arraySize];
```

Alternatively we can create arrays as follows:

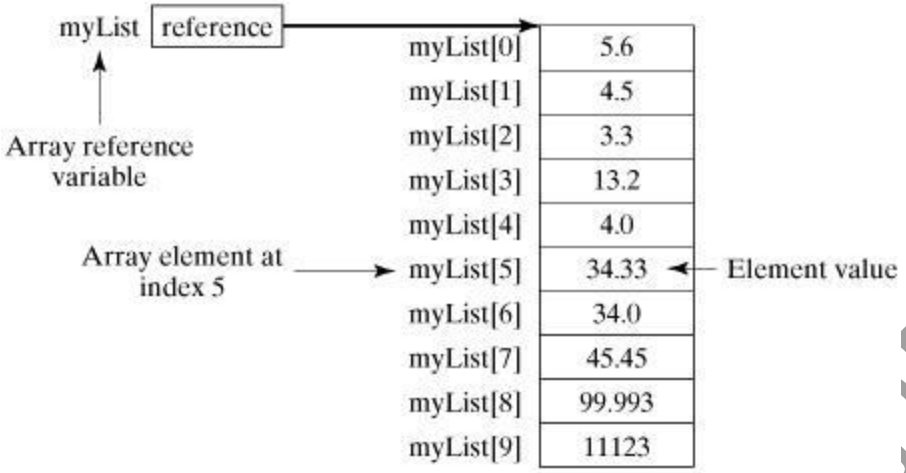
```
datatype[] arrayname = {value0, value1, ..., valuek};
```

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

**Example:** Following statement declares an array variable, a, creates an array of 10 elements of double type, and assigns its reference to a:

```
double[] a = new double[10];
```

Following picture represents array a, which holds ten double values from 0 to




3. **Processing Arrays:** When processing array elements, we often use loop because all of the elements in an array are of the same type and the size of the array is known.

**Example:**

```
class  Student
{
   public static void main(String args[ ])
{
       int marks[ ] = new int[6];
       marks[0] = 50;
       marks[1] = 55;
       marks[2] = 43;
       marks[3] = 61;
       marks[4] = 77;
       marks[5] = 58;
       System.out.println("The marks scored in subject1 is :"+marks[0]);
   }
```

An array initializer is a list of comma separated expressions surrounded by curly braces. The commas separated the values of the array elements. The array will be created large enough to hold exactly the number of elements we specify in the array initializer. **Java strictly checks that we don't store or refer values outside the range of the array.** Java runtime will check to be sure that all array indices are in the correct range. Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray
 {
   public static void main(String[] args)
{
     double[] a = {1.9, 2.9, 3.4, 3.5};
     for (int i = 0; i<a.length; i++)
          System.out.println(a[i] + " ");
     double total = 0;
     for (int i = 0; i < a.length; i++)
         total += a[i];
```

```
        System.out.println("Total is " + total);
        double max = a[0];
        for (int i = 1; i < a.length; i++)
        {
          if (a[i] > max)
                max = a[i];
        }
        System.out.println("Max is " + max);
      }
    }
```

This would produce following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

**Passing Arrays to Methods:** Just as we can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

```
public static void printArray(int[] array)
{
 for (int i = 0; i < array.length; i++)
   System.out.print(array[i] + " ");
 }
```

We can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

**Returning an Array from a Method:** A method may also return an array. For example, the method has shown below returns an array that is the reversal of another array:

```
public static int[] reverse(int[] list)
{  int[] result = new int[list.length];
  for (int i = 0; i = result.length - 1;i <  list.length; i++, j--)
       result[j] = list[i];
  return result;
}
```

**Multi- dimensional arrays:** These are arrays within arrays. The multidimensional arrays are allocated in blocks. If it is an X by Y by Z three-dimensional matrix, then the storage required will be X times Y times Z times the size of the type stored in each cell. In Java we can declare a variable to be three- dimensional but leave out the second and the third dimension then allocate the Y and Z directions separately.

**Eg** : double matrix [ ] [ ] = new double [4] [4];

```
        matrix[0] = new double[4];
        matrix[1] = new double[4];
        matrix[2] = new double[4];
        matrix[3] = {0,1,2,3};
```

**Eg :**

```
class Exarray
{
  public static void main(String args[ ])
{
     int twodim = new int [4] [5];
     int i, j, k = 0;
     for(i = 0; i < 4 ; i++ )
     {
         for(j = 0 ; j<5; j++)
              {
               twodim[i][j] = k ;
               k++;
               }
     }
      for(i = 0; i < 4 ; i++ )
     {
          for(j = 0 ; j<5; j++)
                System.out.println( twodim[i][j] ) ;
      }
   }
}
```

## OPERATORS IN JAVA

Operator is the symbol that tells the computer to perform mathematical or logical manipulations. Java has a rich set of operators as explained below.

1. **The Arithmetic Operators:** Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators. Assume integer variable A holds 10 and variable B holds 20 then:

| Operator | Description | Example |
|---|---|---|
| + | Addition - Adds values on either side of the operator | A + B will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | A - B will give -10 |
| * | Multiplication - Multiplies values on either side of the operator | A * B will give 200 |
| / | Division - Divides left hand operand by right hand operand | B / A will give 2 |

| | | |
|---|---|---|
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | B % A will give 0 |
| ++ | Increment - Increase the value of operand by 1 | B++ gives 21 |
| -- | Decrement - Decrease the value of operand by 1 | B-- gives 19 |

2. **The Relational Operators:** The following are relational operators supported by Java language. Assume variable A holds 10 and variable B holds 20 then:

| Operator | Description | Example |
|---|---|---|
| == | Checks if the value of two operands is equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the value of two operands is equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

3. **The Bitwise Operators:** Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte. Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

a = 0011 1100
b = 0000 1101
---

a&b = 0000 1100
a|b = 0011 1101

a^b = 0011 0001
~a = 1100 0011
The following table lists the bitwise operators. Assume integer variable A holds 60 and variable B holds 13 then:

| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in eather operand. | (A \| B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the efect of 'flipping' bits. | (~A ) will give -60 which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

4. **The Logical Operators:** The following table lists the logical operators. Assume Boolean variables A holds true and variable B holds false then:

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non zero then then condition becomes true. | (A && B) is false. |
| \|\| | Called Logical OR Operator. If any of the two operands are non zero then then | (A \|\| B) is true. |

| | | |
|---|---|---|
| | condition becomes true. | |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

5. **The Assignment Operators**: The assignment operators supported by Java language are:

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |

| | | |
|---|---|---|
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| \|= | bitwise inclusive OR and assignment operator | C \|= 2 is same as C = C \| 2 |

6. **Conditional 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 );
  }
}
```

This would produce following result:

```
Value of b is : 30
Value of b is : 20
```

**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: For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7. Here operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

| 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 |
| Comma | , | Left to right |

## JAVA STATEMENTS

The types of Java Statements are

- **Expression statements** (x=10;)
- **Labeled statements** (goto END; END:.....)
- **Control statements** (if,switch,loops)
- **Synchronization statements** (Threads)
- **Guarding statements** (Exception Handling)

## CONTROL STATEMENTS IN JAVA

Control statements are used to alter the flow of execution according the changes to the state of a program. The control statements are basically categorized into two types.

1. Selection statements
2. Iteration statements

## SELECTION STATEMENTS

There are two types of decision making statements in Java. They are:

- if statements
- switch statements
- jump statements

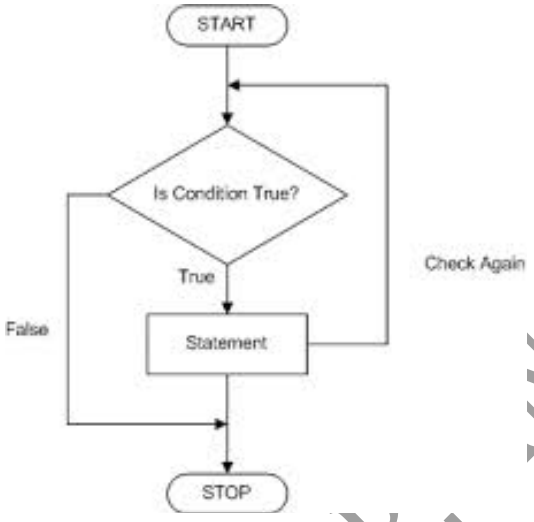
**1.1 The if Statement**: An if statement consists of a Boolean expression followed by one or more statements.

**Syntax:** The syntax of an if statement is:

```
if(Boolean_expression)
{
  //Statements will execute if the Boolean expression
```

```
is true
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.




**Example:**

```
public class Test
{
  public static void main(String args[])
{
    int x = 10;
    if( x < 20 )
       System.out.print("This is if
statement");
   }
}
```

This would produce following result:

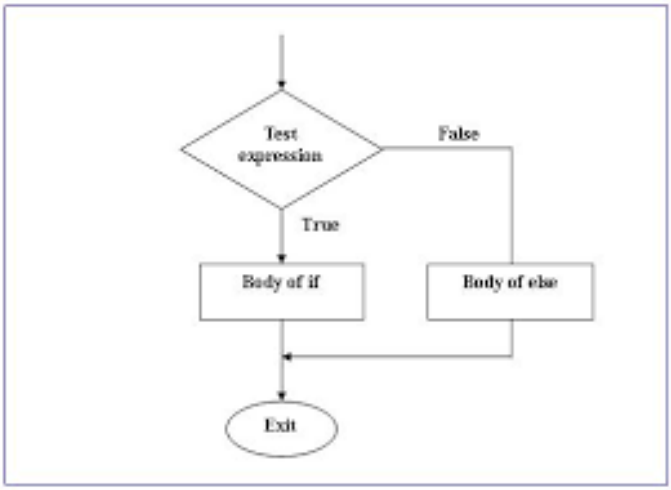
```
This is if statement
```

**1.2 The if...else Statement:** An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

**Syntax:** The syntax of if..else block is,

```
if(Boolean_expression)
{
  //Executes when the Boolean expression is
true
}
else
{
```

```
  //Executes when the Boolean expression is
false
}
```




**Example:**

```
public class Test
 {
  public static void main(String args[])
{
    int x = 30;
    if( x < 20 )
      System.out.print("This is if statement");
    else
      System.out.print("This is else statement");
 }
}
```

This would produce following result:

```
This is else statement
```

**1.3 The Else if ladder Statement:** An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

**Syntax:** The syntax of a if...else is:

```
if(Boolean_expression 1)
{
  //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2)
{
```

```
  //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3)
{
  //Executes when the Boolean expression 3 is true
}else
{
  //Executes when none of the above condition is
true.
}
```

**Example:**

```
public class Test
{  public static void main(String args[])
{
    int x = 30;
    if( x == 10 )
      System.out.print("Value of X is 10");
    else if( x == 20 )
      System.out.print("Value of X is 20");
    else if( x == 30 )
      System.out.print("Value of X is 30");
    else
      System.out.print("This is else statement");
}
}
```

This would produce following result:

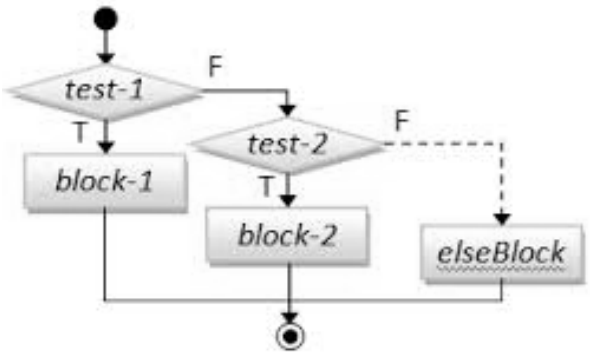
```
Value of X is 30
```

**1.4 Nested if Statement:** It is always legal to nest if-else statements, which means we can use one if or else if statement inside another if or else if statement.

**Syntax:** The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1)
{
  //Executes when the Boolean expression 1 is true
  if(Boolean_expression 2)
  {
    //Executes when the Boolean expression 2 is
true
  }
}
```

We can nest *else if...else* in the similar way as we have nested *if* statement.




**Example:**

```
public class Test
{
  public static void main(String args[])
{
    int x = 30;   int y = 10;
    if( x == 30 )
  {
      if( y == 10 )
        System.out.print("X = 30 and Y = 10");
  }
}
}
```

This would produce following result:

```
X = 30 and Y = 10
```

**1.5 The switch Statement:** A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

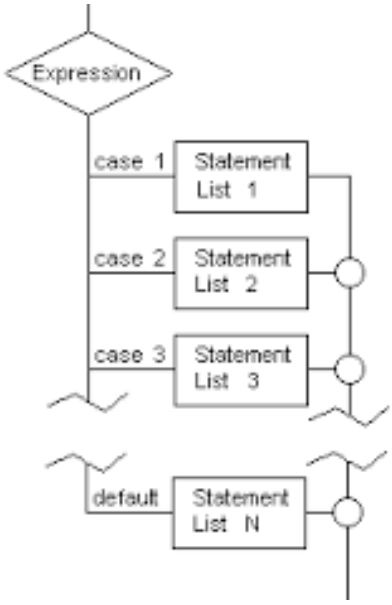
**Syntax:** The syntax of enhanced for loop is:

```
switch(expression)
{
   case value :
     //Statements
     break;
   case value :
     //Statements
     break;
   //We can have any number of case
statements.
   default : //Optional
     //Statements
}
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.

- We can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.




**Example:**

```
public class Test
 {
  public static void main(String args[])
{
    char grade = args[0].charAt(0);
    switch(grade)
    {
    case 'A' :
      System.out.println("Excellent!");
      break;
    case 'B' :
    case 'C' :
      System.out.println("Well done");
      break;
    case 'D' :
```

```
            System.out.println("We passed");
          case 'F' :
            System.out.println("Better try again");
            break;
          default :
            System.out.println("Invalid grade");
          }
          System.out.println("Grade is " + grade);
        }
      }
```

Compile and run above program using various command line arguments. This would produce following result:

```
$ java Test a
Invalid grade
Grade is a a
$ java Test A
Excellent!
Grade is a A
$ java Test C
Well done
Grade is a C
$
```

**ITERATIVE STATEMENTS**

Java has very flexible three looping mechanisms. We can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

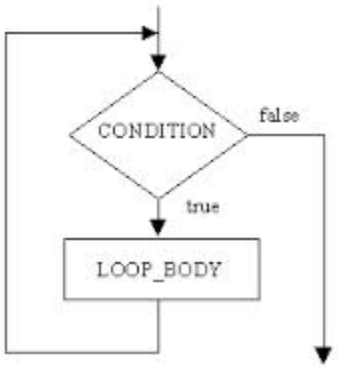
As of java 5 the *enhanced for loop* was introduced. This is mainly used for Arrays.

**2.1 The while Loop:** A while loop is a control structure that allows repeating a task a certain number of times.

**Syntax:** The syntax of a while loop is:

```
while(Boolean_expression)
{
  //Statements
}
```

When executing, if the *boolean_expression* result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true. Here key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.




**Example:**

```
public class Test
 {
   public static void main(String args[])
{    int x= 10;
      while( x < 20 )
     {
        System.out.print("value of x : " + x );
        x++;
        System.out.print("\n");
      }
   }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

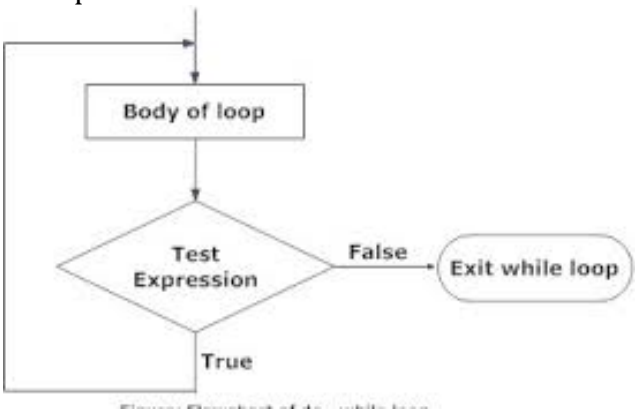
**2.2 The do...while Loop:** A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

**Syntax:** The syntax of a do...while loop is:

```
do
{
  //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested. If the Boolean expression is true, the flow of

control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.




**Example:**

```
public class Test
{
  public static void main(String args[])
{
     int x= 10;
     do
    {
       System.out.print("value of x : " + x );
       x++;
       System.out.print("\n");
     }while( x < 20 );
  }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

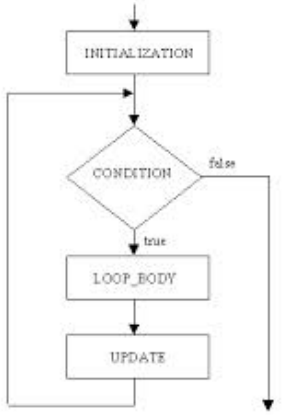
**2.3 The for Loop:** A for loop is a repetition control structure that allows we to efficiently write a loop that needs to execute a specific number of times. A for loop is useful when we know how many times a task is to be repeated.

**Syntax:** The syntax of for loop is:

```
for(initialization; Boolean_expression; update)
{
        //Statements
}
```

Here is the flow of control in for loop:

1. The initialization step is executed first, and only once. This step allows we to declare and initialize any loop control variables. We are not required to put a statement here, as long as a semicolon appears.
2. Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past for loop.
3. After the body of for loop executes the flow of control jumps back up to the update statement. This statement allows we to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
4. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, for loop terminates.




**Example:**

```
public class Test
{
  public static void main(String args[])
{
     for(int x = 10; x < 20; x = x+1)
    {
       System.out.print("value of x : " + x );
       System.out.print("\n");
     }
  }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

## JUMP STATEMENTS

1. **The break Keyword:** The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement. The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

**Syntax:** The syntax of a break is a single statement inside any loop:

```
break;
```

**Example:**

```
public class Test
{
  public static void main(String args[])
{
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x=1;x<numbers.length;x++)
    {
      if( x == 30 )
          break;
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This would produce following result:

```
10
20
```

2. **The continue Keyword:** The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.
   - In for loop, the continue keyword causes flow of control to immediately jump to the update statement.
   - In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

**Syntax:** The syntax of continue is a single statement inside any loop:

```
continue;
```

**Example:**

```
public class Test
 {
   public static void main(String args[])
{
     int [] numbers = {10, 20, 30, 40, 50};
     for(int x=1;x<numbers.length;x++)
  {
        if( x == 30 )
            continue;
        System.out.print( x );
        System.out.print("\n");
     }
   }
}
```

This would produce following result:

```
10
20
40
50
```



## Sample Programs

| | |
|---|---|
| Arithmetic Operators | Biggest/Smallest element in the given array |
| Biggest/Smallest of 3 numbers using ?: | Sorting of numbers in ascending & descending order |
| Simple marklist | Odd/Even/.Positive/Negative number |
| Mathematical functions | Prime number |
| Sum of elements in an array | Perfect number |
| Pascal triangle (for loop) | Menu using do..while |
| Sum of digits | Reversing a number |
| Printing a number in words | Multiplication table |
| Linear search | Addition/Subtraction of 2 matrices |
| Area and Perimeter of shapes | Copying of an array into another |
| Sum of two arrays | Merging two arrays |
| Reverse an array | Binary search |

## UNIT 2

**Classes - Objects - Constructors - Overloading method - Access Control- Static and Final methods - Inner Classes - String Objects - String Class -String Buffer - Char Array - Java Utilities - Inheritance - Overriding methods - Using super-Abstract class.**

## CLASSES AND OBJECTS

**Classes – Declaration:** A class is a logical construct upon which objects can be built. The class can be considered as a template for objects. All our applications are encapsulated within a class. We can say that class is a new type of data defined by the user. The general form of declaration of a class is as follows:

```
class Classname
{
      datatype instancevariable1;
      datatype instancevariable2;
       ……….……
      returntype  methodname1( list of parameters)
        {
            method – implementation ;
       }
      returntype methodname2 (list of parameters)
        {
           method- implementation;
       }
     ……..……..
}
```

The class declaration starts with the keyword “class” followed by the name of the class which the user can give with relevance to his application. The scope of the class lies between the curly braces following the class name and the braces after the method declaration.

In the declaration we have specified the instance variables inside the class. These are variables which belong to every instance or object of a class. Hence they are given the name instance variables. In addition to instance variables, we can also declare class variables. Hope we remember that the keyword ‘static’ used in the main method specifies that main is a class method. So **static** is the access modifier that specifies that the data member or a method is a class variable or class method.

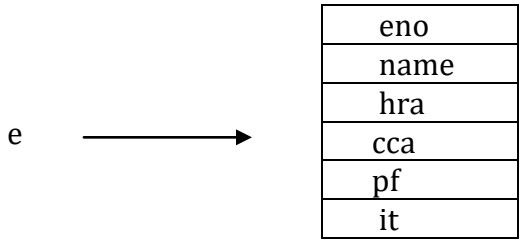
The method definition follows the declaration of the variables. If our method returns some value, specify the return type which is followed by the method name.Let us declare a simple class called Employee.

```
class Employee
{
      int eno;
      String ename;
      double hra;
```

```
    double cca;
    double pf;
    double it;
};
```

**Objects – Declaration:** As we know the class is a new type of data declared by the user. This type can be used to declare objects of the same type. For ex, if we want to create an instance of type Employee, the statement is

Employee e = new Employee(); where we are declaring a variable 'e' of type employee. The new operator allocates memory for Employee object at runtime and reference is assigned to 'e'. So 'e' can be called as object reference variable.

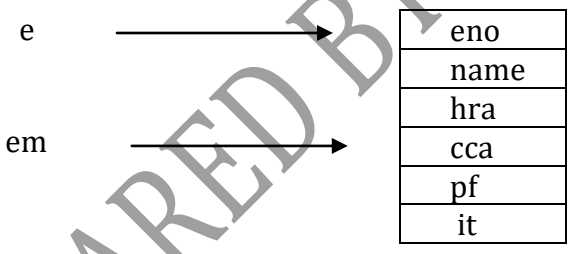



The object reference variables behave differently when an assignment is made . For ex ,

Employee e = new Employee();

Employee em = e ;

**e, em** doesn't refer to two different objects because we cannot allocate memory for employee object in the second statement and **em** just refers to the same object which **e** is referring.




**new operator:** The new operator dynamically allocates memory for an object. It has the general syntax :

class var = new classname();

class-var is the variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what happens when an object of a class is created. If no explicit constructor is defined, then the compiler will automatically invoke a default constructor.

Since the memory allocation takes place at runtime, this approach is advantageous. Like, our program can create as many as objects as we need during the execution of the program. As memory is finite, there is a possibility that **new** will not be able to allocate memory for an object when memory is insufficient. A run-time exception will occur. We can handle such errors using exception handling.

**Methods:** Although it is perfectly fine to create a class that contains only data, it rarely happens. We will use methods to access the instance variables. Let us use methods to print the net salary of the Employee class which we have declared earlier.

```
class Employee
{
 int eno;
 String ename;
 double hra;
 double cca;
double pf; double it;
 public void netsal()
        {
                System.out.println((hra+cca) –( pf + it));
         }
}
public class Empex
{
public static void main(String args[ ] )
{
  Employee  e = new Employee();
  e.eno = 1;
  e.ename = “Asha”;
  e.hra = 200;
  e.cca = 150;
  e.pf = 100;
  e.it = 150;
  e.netsal();
}
}
```

When e.netsal() is invoked, the Java run-time system transfers control to the code defined inside netsal(). After the statements inside netsal() have executed, control is returned to the calling routine . The instance variables pf, it , hra and cca are referred directly without using dot operator. When a method uses an instance variable defined inside a class it can use it directly without explicit reference to an object and without using the dot operator. A method is invoked relative to some object of the class. Thus there is no need to specify the object the second time within the method.

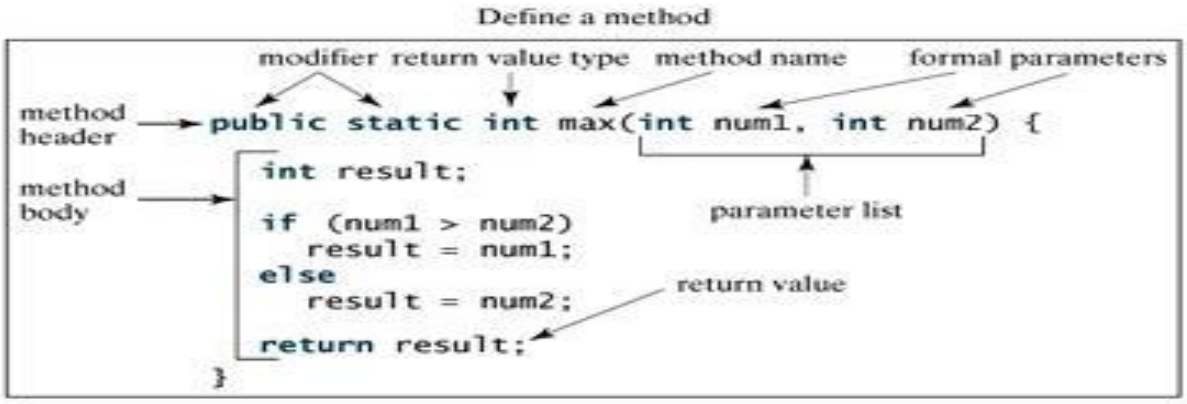
In general, a method has the following syntax:

```
modifier returntype methodname(list of parameters)
{
  // Method body;
}
```

A method definition consists of a method header and a method body. Here are all the parts of a method:

- **Modifiers:** The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.

- **Return Type:** A method may return a value. The return type is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the return type is the keyword **void**.
- **Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature.
- **Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method Body:** The method body contains a collection of statements that define what the method does.




**Example:** Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** Return the max between two numbers */
public static int max(int num1, int num2)
 {
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

**Calling a Method:** To use a method, we have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not. When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached. If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(30, 40);
```

If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

**Example:**

Following is the example to demonstrate how to define a method and how to call it:

```
public class TestMax
 {
   public static void main(String[] args)
{
     int i = 5;
     int j = 2;
     int k = max(i, j);
     System.out.println("The maximum between " + i +" and " + j + " is " + k);
}
/** Return the max between two numbers */
public static int max(int num1, int num2)
 {
  int result;
  if (num1 > num2)
     result = num1;
  else
     result = num2;
  return result;
}
```

This would produce following result:

```
The maximum between 5 and 2 is 5
```

This program contains the main method and the max method. The main method is just like any other method except that it is invoked by the JVM. The main method's header is always the same, like the one in this example, with the modifiers public and static, return value type void, method name main, and a parameter of the String[] type. String[] indicates that the parameter is an array of String.

**The void Keyword:** This section shows how to declare and invoke a void method. Following example gives a program that declares a method named printGrade and invokes it to print the grade for a given score.

**Example:**

```
public class TestVoidMethod
{
  public static void main(String[] args)
 {
    printGrade(78.5);
  }
  public static void printGrade(double score)
```

```
 {
    if (score >= 90.0)
       System.out.println('A');
    else if (score >= 80.0)
      System.out.println('B');
    else if (score >= 70.0)
      System.out.println('C');
    else if (score >= 60.0)
      System.out.println('D');
    else
      System.out.println('F');
      }
}
```

This would produce following result:

```
C
```

Here the printGrade method is a void method. It does not return any value. A call to a void method must be a statement. So, it is invoked as a statement in line 3 in the main method. This statement is like any Java statement terminated with a semicolon.

**Passing Parameters by Values:** When calling a method, we need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association. For example, the following method prints message n times:

```
public static void show(String message, int n)
{
for (int i = 0; i < n; i++)
   System.out.println(message);
}
```

Here, we can use show("Hello", 3) to print "Hello" three times. The show("Hello", 3) statement passes the actual string parameter, "Hello", to the parameter, message; passes 3 to n; and prints "Hello" three times. However, the statement show(3, "Hello") would be wrong. When we invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method. For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means passing the value of x to y.

**Example:** Following is a program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The swap method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

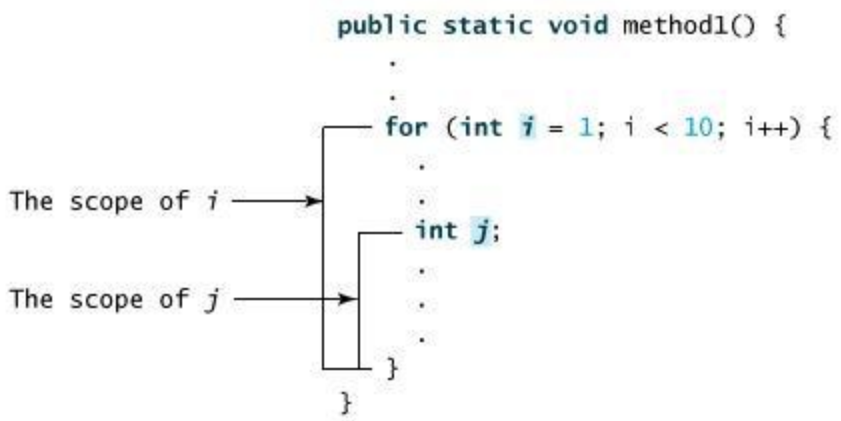
```
public class TestPassByValue
{
  public static void main(String[] args)
{
    int num1 = 1;
    int num2 = 2;
    System.out.println("Before swap method, num1 is " +num1 + " and num2 is " + num2);
    // Invoke the swap method
    swap(num1, num2);
    System.out.println("After swap method, num1 is " +
                num1 + " and num2 is " + num2);
  }
  /** Method to swap two variables */
  public static void swap(int n1, int n2)
{
    System.out.println("\tInside the swap method");
    System.out.println("\t\tBefore swapping n1 is " + n1 + " n2 is " + n2);
    // Swap n1 with n2
    int temp = n1;
    n1 = n2;
    n2 = temp;
    System.out.println("\t\tAfter swapping n1 is " + n1 + " n2 is " + n2);
  }
}
```

This would produce following result:

```
Before swap method, num1 is 1 and num2 is 2
 Inside the swap method
            Before swapping n1 is 1 n2 is 2
            After swapping n1 is 2 n2 is 1
After swap method, num1 is 1 and num2 is 2
```



**The Scope of Variables:** The scope of a variable is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used. A parameter is actually a local variable. The scope of a method parameter covers the entire method. A variable declared in the initial action part of for loop header has its scope in the entire loop. But a variable declared inside for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable as shown below:




We can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but we cannot declare a local variable twice in nested blocks.

**Methods returning a value:** We must consider certain features with these methods returning a value. i. e.,

1. The type of data returned by a method must be compatible with the return type specified by the method.
2. The variable receiving the value returned by a method must also be compatible with the return type specified for the method

```
class Student
{
      int  rno;
      String sname;
       int m1;
       int m2;
       public int rettotal()
          {
           return (m1+m2);
      }
}

public class Studex
{
   public static void main(String args [ ] )
{
    Student s = new Student();
     s.rno = 5;
     s.sname = "Swathee";
     s.m1 = 86;
     s.m2 = 89;
    System.out.println( "RNo Name   Mark1  Mark2 Total" );
    System.out.println( s.rno+" "+s.sname+" "+s.m1+"   "+s.m2+"    "+s.rettotal());
   }
}
```

The method rettotal() returns the total marks of the student. The first println statement prints the heading.The second println statement prints the value for rollno, name of the student , his/her marks and the total marks. The blank space in between the values is provided by the space between the double quotes in the second print statement. i.e s.rno+ followed by double quotes and few blank spaces and again double quotes provides the required blank space between s.rno and s.sname. Similarly we have provided blank spaces between the other values. In this statement + is the operator that is used for concatenation.

**Adding methods that takes parameters:** In the above examples, we have assigned the values for instance variables as follows ,

s. rno = 5;

s.sname = “Swathee”; etc. Sometimes with this type of assignment we might forget to give some values. Moreover, in well defined programs, instance variables should be accessed only through methods.

```
class Student
{
          int  rno;
          String sname;
          int m1;
          int m2;
          public void setvalues(int rollno, String name, int mark1, int mark2)
        {
            rno = rollno;
            sname = name;
            m1 = mark1;
             m2 = mark2;
         }
          public int rettotal()
          {
           return (m1+m2);
         }
}
public class Studex
{
   public static void main(String args [ ] )
  {
     Student s = new Student();
     s.setvalues(1, “Sruthee”, 78, 90);
    System.out.println( "RNo Name   Mark1  Mark2 Total" );
    System.out.println(s.rno+" "+s.sname+" "+s.m1+"   "+s.m2+"     "+s.rettotal());
   }
}
```

**Using Command-Line Arguments:** Sometimes we want to pass information into a program while execution. This is accomplished by passing command-line arguments to

main( ). A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

**Example:** The following program displays all of the command-line arguments that it is called with:

```
class CommandLine
{
  public static void main(String args[])
{
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine this is a command line 200 -100
```

This would produce following result:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

## CONSTRUCTORS

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. Typically, we will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object. All classes have constructors, whether we define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once we define our own constructor, the default constructor is no longer used.

**Example:**

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass
{
  int x;
```

```
  MyClass()
 {  x = 10;  }
}
```

We would call constructor to initialize objects as follows:

```
class ConsDemo
 {  public static void main(String args[])
 {    MyClass t1 = new MyClass();
     MyClass t2 = new MyClass();
     System.out.println(t1.x + " " + t2.x);
  }
}
```

Most often we will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method- just declare them inside the parentheses after the constructor's name.

**Example:** Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass
 {
  int x;
  MyClass(int i )
 {    x = i;  }
}
```

We would call constructor to initialize objects as follows:

```
class ConsDemo
{
  public static void main(String args[])
 {
    MyClass t1 = new MyClass( 10 );
    MyClass t2 = new MyClass( 20 );
    System.out.println(t1.x + " " + t2.x);
  }
}
```

This would produce following result:

```
10 20
```

When we create an object of a class, a special kind of method called a **constructor** is always invoked. If we don't define a constructor the compiler will provide a default constructor. The constructor's main purpose is to initialize the instance variables. The two main features that distinguish constructors from other class methods are

1. A constructor never returns any value and we needn't specify the return type
2. The constructor has the same name as the class.

```
class Box
{
      double width;
      double height;
      double depth;
      Box ()
          {
           width = 15.5;
           height = 10.3;
           depth = 12.7;
      }
     public double vol()
   {  return (width * height * depth);  }
}
public class Boxex
{
   public static void main(String args[ ])
{
     Box  b1 = new Box();
     Box  b2 = new Box();
    System.out.println("Volume of the box "+b1.vol());
    System.out.println("Volume of the box "+b2.vol());
   }
}
```

**Parameterized constructors:** We have used a constructor, where the dimensions are similar for all box objects that are created. This may not be of much use for us. We need to create box objects of different dimensions. This can be done by adding parameters to constructors.

**Example:**

```
class Box
{
      double width;
      double height;
      double depth;
      Box (double w, double h, double d)
          {
           width = w;
           height = h;
           depth = d;
      }
     public double vol()
          {  return (width * height * depth);  }
}
```

```
public class Boxex
{
   public static void main(String args [ ] )
{
     Box  b1 = new Box(10.54, 12.89, 10.43);
     Box b2 = new Box(12.5, 13, 15.8);
    System.out.println("Volume of box1 "+b1.vol());
    System.out.println("Volume of box2 "+b2.vol());
   }
}
```

**Copy constructors:** This is used when one object's data has to be copied to another object.

**Eg:**

```
Box b2=new Box(b1);
Box(Box b)
{
        width=b.width;
        height=b.height;
        depth=b.depth;
}
```

**this KEYWORD: this** keyword can be used inside any method to refer the current object. **this** is a reference to the object on which the method was invoked.

**Instance variable hiding:** It is illegal to have two local variables with the same name within the same scope. But we can have a local variable with the same name as that of a instance variable. In that case, the local variable hides the instance variable, and a name collision occurs as both these have the same name. Such name collisions can be resolved with the help of this keyword. In the above example, we have the formal parameters w, h and d for the constructor Box. If we have the name for these parameters as width, height and depth, then our constructor changes with the use of this keyword as follows,

```
Box (double width, double height, double depth)
    {
     this.width = width;
     this.height = height;
     this.depth = depth;
}
```

## METHOD OVERLOADING

The Java compiler determines which method is used based on the method signature. Overloading methods can make programs clearer and more readable. Methods that perform closely related tasks should be given the same name.

Overloaded methods must have different parameter lists. We cannot overload methods based on different modifiers or return types. Sometimes there are two or more possible matches for an invocation of a method due to similar method signature, so the

compiler cannot determine the most specific match. This is referred to as ambiguous invocation.

Java allows us to define several methods in a class with the same name as long as each method has a set of parameters that is unique. This is called as **method overloading.** The name of a method with the type and number of parameters form the **signature** of the method. Method overloading is one way through which Java implements polymorphism. The overloaded methods must differ in the type and /or the number of parameters. They may have different return types but the difference in return type alone is not sufficient to distinguish two versions of a method.

```
class  Add
{
   void sum(int a, int b)
   {
            System.out.println(" the result is  :" +(a+b));
    }
   void sum(double a, double b)
  {
         System.out.println(" the result  is : "+ (a+b));
    }
}

class Addex
{
   public static void main(String args [ ])
{
         Add a1 = new Add();
         Add a2 = new Add();
         a1.sum(10,15);
         a2.sum(10.9,23.67);
    }
}
```

In the above example the method sum() is overloaded. The type of parameter passed to the first version is int, and then we have passed values of type double. When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. Then the corresponding method is invoked. The max method that was used earlier works only with the int data type. But what if we need to find which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2)
{
 if (num1 > num2)
   return num1;
 else
```

```
   return num2;
}
```

If we call max with int parameters, the max method that expects int parameters will be invoked; if we call max with double parameters, the max method that expects double parameters will be invoked. This is referred to as **method overloading**; that is, two methods have the same name but different parameter lists within one class.

**Overloading constructors:** Similar to method overloading we can overload constructors. In our example Box we have used constructor with three parameters. Every time we need to pass three arguments to this constructor. If we don't provide the three arguments then we will get an error. Sometimes we might need to create objects and we need to initialize all the dimensions with a same value. We can do this with the help of constructor overloading.

**Example:**

```
class Box
{
      double width;
      double height;
      double depth;
      Box (double w, double h, double d)
          {
           width = w;
           height = h;
           depth = d;
      }
      Box(double val)
          {    width = height = depth = val;    }
     public double vol()
          {   return (width * height * depth); }
}
```

```
public class Boxex
{
   public static void main(String args[ ] )
{
     Box  b1 = new Box(10, 12, 10);
     Box b2 = new Box(12.5);
    System.out.println("Volume of box1"+b1.vol());
    System.out.println("Volume of box2: " +b2.vol());
   }
}
```

So the second version of the constructor passes a value 12.5 which is assigned to all the three dimensions width, height and depth.

## STATIC KEYWORD

"Common to all objects and accessed without using a particular object".

**Eg:**

```
static int x;                    //class variables
static float mul(int x);                 //class methods
double x=Math.sqrt(10);     //sqrt is static and so it is called using classname.
```

**Eg:**

```
class Operation
{
static int mul(int x,int y)
{return (x*y);}
static int div(int x,int y)
{return (x/y);}
class MainClass
{
public static void main(String args[])
{
int a=Operation.mul(3,2);
int b=Operation.div(a,2);
System.out.println(b);
}
}
```

**Rules for static methods:**

- Can access only static variables.
- Can call other static methods only.
- Cannot refer to this or super keyword in any way.

## FINAL KEYWORD

Java's final keyword is similar to const keyword in C++. When variables are declared with the access modifier **final,** its value cannot be changed. "This cannot be changed"-Final. **final int a = 3;** Here the value of a remains constant and it cannot be modified as it has been declared as final. Three uses of final keyword are:

1. **Final data**: Here the data is treated as constant. A field is both static and final has only one piece of storage that cannot be changed. A final data can be initialized at the time of declaration itself or afterwards when its actual usage arises (blank final).
2. **Final methods**: There are two reasons for final methods. The first is to put a "lock" on the method to prevent any inheriting class from changing its meaning. The second reason for final methods is efficiency-the compiler is allowed to turn any calls to that method into inline calls. It is noted that any private methods in a class is implicitly final.
3. **Final classes**: When an entire class is declared final, no class is allowed to inherit from the final class. The final class is designed in such a way that the classes derived from it can never make any changes for security reasons.

**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. Java calls that method whenever it is about to reclaim the space for that object. This is similar to destructor in C++. It is just opposite to initialization. Add keyword finalize () to the methods and define the tasks to be done explicitly. This method is called finalize( ), and it can be used to ensure that an object terminates cleanly. For example, we might use finalize( ) to make sure that an open file owned by that object is closed.To add a finalizer to a class, we 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 we 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 we cannot know when.or even if.finalize( ) will be executed. For example, if our program ends before garbage collection occurs, finalize( ) will not execute.

Methods can also be declared as final. In that case these methods cannot be overridden. So when we want to avoid certain methods from being overridden we can declare them as final. Methods declared as final can sometimes provide a performance enhancement. The compiler is free to inline calls because it knows they will not be overridden by a subclass. Since final methods cannot be overridden, a call to such a method can be resolved at compile time . This is called as **early binding.** Whereas when methods are overridden , a call to such method is resolved only at runtime which is called as **late binding**.(ie dynamic method dispatch). Usage of final modifier in a class prevents it from being inherited.

```
final class A
{
  int a = 10;
  void display()
{
    System.out.println("value of a"+a);
  }
}
class B extends A
{     // not possible as class A is declared as final
  ----------------------
}
```

When a class is declared as final by default, all the methods in that class are final. So the methods inside a final class cannot be overridden.

**Points on Keyword final:**

1. Final keyword can be applied to variable, method or class.

2. We cannot reassign the value of final variable (It can be blankfinal-declared as final but not initialized).
3. Use capital letters to differentiate final identifiers from others.
4. Final method cannot be overridden.
5. Final class cannot be inheritable.
6. Final is different from keyword finally and finalize() method.
7. Java interface variables are implicitly final.
8. Final and abstract are two opposite keyword and a final class cannot be abstract in Java.
9. Final methods are bonded during compile time.
10. Making a class, method or variable as final helps to improve performance because JVM gets an opportunity for assumption and optimization.

## INHERITANCE

Defining a new class based on an existing class is called **derivation.** The new class or derived class is called as the direct **subclass** of the class from which it is being derived. The original class is called as the base class as it forms the base for the definition of the derived class. The base class can also be called as **super class.**

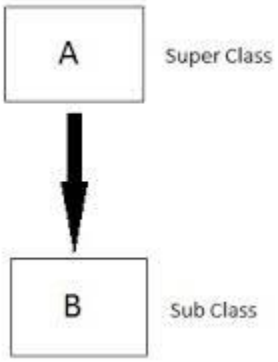
The inclusion of members of a base class in a derived class so that they are accessible in the derived class is called **class Inheritance.** If a base class member is not accessible in a derived class, then it is not an inherited member of the derived class. Objects of the derived class type will contain all the inherited members of the base class – both fields and methods.

Java uses extends keyword instead of the ":" in C++ inheritance. All inheritance in Java is public and there is no protected or private inheritance. Let us see a simple example for inheritance. A Son may have certain characteristics resembling his Father. In addition to that he will have his unique features. So we can say that the Son has derived certain features from his father which implies that Son can be called as a subclass of the superclass Father. The keyword 'extends' is used to extend the properties of superclass to its subclass. Now the subclass has its own properties+its own methods+superclass properties+superclass methods. The general declaration for a class that inherits a superclass is

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

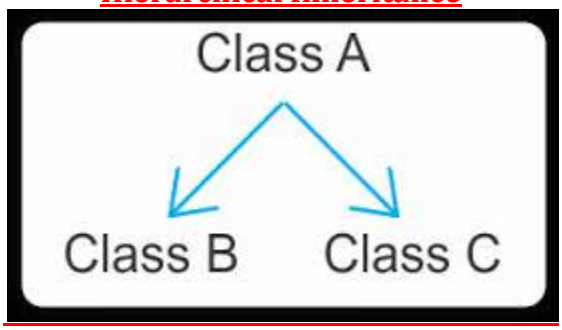
Java doesn't support multiple inheritance (not directly but by the concept of interface), but supports single, multi-level, hierarchical and hybrid inheritance. So we can specify only one super class for any sub class we create. With multi-level inheritance, we can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass.

## Single Inheritance




```
public class Room1
{
int l,b;
Room1(int x,int y)
{l=x;b=y;}
int area()
{return(l*b);}
}
public class Room2 extends Room1
{
int h;
Room2(int x,int y,int z)
{super(x,y);h=z;}
int vol()
{return(l*b*h);}
}
class MainClass
{
public static void main(String args[])
{
        Room2 r=new Room2(8,10,12);
        System.out.println(r.area());
        System.out.println(r.vol());
}
}
```

## Hierarchical Inheritance



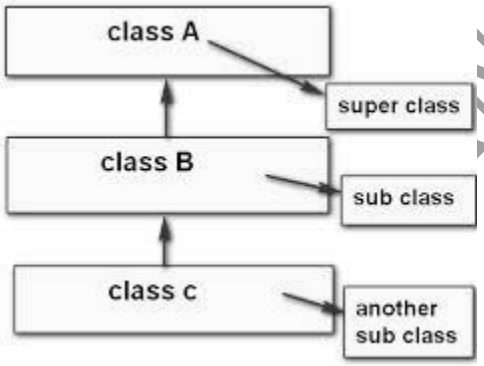

It refers to the existence of many subclasses from a super class.

```
class Shape
 {
  int d1;
  int d2;
  public void setvalues(int dimen1, int dimen2 )
{
    d1 = dimen1;
    d2 = dimen2;
  }
}
class Triangle extends Shape
 {
  int ar ;
  public void  area()
{
    ar = (d1 * d2 )/ 2;
    System.out.println("Area of triangle:"+ar);
  }
}
class Rectangle extends Shape
{
   int ar;
   public void area()
{
    ar = d1* d2;
    System.out.println("Area of rectangle :"+ar);
   }
}
public class Inher
{
 public static void main(String args [ ])
{
     Triangle t = new Triangle();
     t.setvalues(10,10);
     Rectangle r = new Rectangle();
```

```
    r.setvalues(5, 10);
    t.area();
    r.area();
 }
}
```

We have defined a class Shape and we have assigned the dimensions. The two subclass inherited from Shape are Rectangle and Triangle. We have a method setvalues() in the base class which can be used to assign the dimensions. This base class method is inherited in both the subclasses. i.e. We have invoked the same method with reference to the object of Triangle, where we pass the values for base and height. And in the case of rectangle the values for breadth and height are passed.

**Multi-level Inheritance**




Class A is the super class of B and class B is the super class of class C. So class B can access the members of A, and class C can access the members of B (of course only the public and protected members), which means that class C can access the members of A.

**Eg 1**:

```
class Class1
{
     int x;
     Class1(int a)
        {
         x=a;
       }


 public void showx()
    {
        System.out.println("x="+x);
    }
}
class Class2 extends Class1
{
    float y;
```

```
     Class2(float b)
          {   y=b;}
      public void showy()
     {
          System.out.println("y="+y);
     }
}

class Class3 extends Class2
{
    float z;
     public void showz()
     {
         showx();showy();z=x+y;
          System.out.println("z="+z);
     }
}
public class MainClass
{
   public static void main(String args [ ] )
{
          Class3 obj=new Class3();
          obj.showz();
   }
}
```

**Eg 2:**

```
// Multilevel Inheritance
import java.util.*;
class Student
{
int rollno;
String name;
void get1()
{
Scanner s=new Scanner(System.in);
System.out.println("Enter the student roll number");
rollno=Integer.parseInt(s.nextLine());
System.out.println("Enter the student name");
name=s.nextLine();
}
}
class Marks extends Student
{
int marks[]=new int[3];
```

```
void get2()
{
Scanner s=new Scanner(System.in);
System.out.println("Enter the 3 marks of the student" );
for(int i=0;i<3;i++)
  marks[i]=Integer.parseInt(s.nextLine());
}
}
class Result extends Marks
{
int total;
String result;
void put()
{
get1();get2();
total=marks[0]+marks[1]+marks[2];
if (marks[0]>=35 && marks[1]>=35 && marks[2]>=35)
   result="pass";
else
   result="fail";
System.out.println("STUDENT'S ROLL NUMBER="+rollno);
System.out.println("STUDENT'S NAME      ="+name.toUpperCase());
System.out.println("STUDENT'S MARK 1    ="+marks[0]);
System.out.println("STUDENT'S MARK 2    ="+marks[1]);
System.out.println("STUDENT'S MARK 3    ="+marks[2]);
System.out.println("STUDENT'S TOTAL     ="+total);
System.out.println("STUDENT'S RESULT    ="+result.toUpperCase());
}
}
public class Multilevel
{
public static void main(String args[])
{
Result r=new Result();
r.put();
}
}
```

## USING Super

It is the duty of the subclass to construct the values of both superclass and subclass. Java provides the way for the subclass to refer its immediate superclass by using the keyword super.

- super is used in the subclass constructor only.
- The call to superclass constructor must appear as the first statement in the subclass constructor.

- Parameters in super() must match the order and data types of the superclass variables.

**super has 2 forms**

1. To access the superclass's constructor
2. To access the member of a superclass

**To access superclass constructor**: A subclass can access the superclass constructor with the following syntax,

**super(parameter-list);**

parameter-list specifies any parameters needed by the constructor in the superclass super() must be the fist statement executed inside a subclass's constructor.

**Eg:**

```
class Shape
{
  int length;
  int breadth;
 Shape(int dimen1, int dimen2 )
{
    length = dimen1;
    breadth = dimen2;
  }
}
class Cuboid extends Shape
{
  int height;
 Cuboid(int dimen1, int dimen2, int dimen3)
 {
    super(dimen1, dimen2);
    height = dimen3;
 }
 public int  volume()
{     return(  length *breadth* height ); }
}
public class Shvolume
{
   public static void main(String args [ ])
{
     Cuboid c = new Cuboid(6, 8 ,7);
     System.out.println("the volume of cuboid :"+c.volume());
 }
}
```

In the above example we have initialized the values for dimensions1 and 2 in the constructor of superclass Shape. And the subclass Cuboid needs one more dimension to calculate its volume which we have initialized in its constructor. The superclass's constructor is called inside the constructor of the subclass with the keyword super().

**To access a superclass member:** The general form is :

**super.member**

member can be a method or a instance variable.

**Eg:**

```
class Univexam
{
  int marks;
}
class  Internal extends Univexam
{
   int marks;
    Internal(int a, int b)
 {
      super.marks = a;
       marks = b;
   }
   void obtained()
 {
     System.out.println("marks in university exam" +super.marks);
     System.out.println("marks in internal exam" +marks);
     System.out.println("Total marks obtained"+(super.marks+marks));
  }
}
class Usesuper
{
  public static void main(String args[ ])
{
    Internal  iobj = new Internal(40, 18);
    iobj.obtained();
  }
}
```

In the above example we have used the same name marks for specifying the internal as well as the external marks. So there is a name collision between the variable in the super class and the sub class. Hence, the instance variable marks in the super class Univexam can be accessed in the subclass with the keyword super. ie super.marks, and the instance variable marks in class Internal can be accessed directly as marks.

## METHOD OVERRIDING

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

**Eg:**

```
class A
{
   int i, j;
  A(int a, int b)
   {
        i = a;
        j = b;
   }
    void show()
   { System.out.println(“ values of i and j : +i + “  “ +j ); }
}
class B extends A
{
    int k;
    B(int a, int b, int c)
   {
       super(a, b);
        k = c;
    }
   void show()
   {
              System.out.println(“value of k:”+k);
    }
}
class Override
 {
   public static void main(String args[ ])
   {
       B ob1 = new B(10,20,30);
        Ob1.show();
    }
 }
```

When show() is invoked on an object of class B, the version of show() defined within B is used. This show() method in B overrides the show() in A.

## ABSTRACT CLASSES AND METHODS

Sometimes we will create a class without giving implementation for all methods. For ex- let us consider the example **Shape.**

**Example:**

```
class Shape
 {
  int d1;
  int d2;
  public void setvalues(int dimen1, int dimen2 )
 {
```

```
    d1 = dimen1;
    d2 = dimen2;
  }
 public void  area()
{
     System.out.println(“Area not defined”);
 }
}
class Triangle extends Shape
 {
  int ar ;
  public void  area()
 {
    ar = (d1 * d2 )/ 2; System.out.println("Area of triangle:"+ar);
  }
}
public class Inher
{
 public static void main(String args [ ])
{
     Shape s  = new Shape();
     Triangle t = new Triangle();
     t.setvalues(10,10);
      t.area();
 }
}
```

In the superclass Shape there is no need to give the implementation for the method area() because it is not meaningful. Only in the subclasses we have a meaningful implementation by calculating the areas of the corresponding to the shapes triangle and rectangle. We can avoid the message in the body of the method area() inside the superclass Shape. When a method doesn't have any implementation for its method, it is said to be an abstract method. It is declared using the abstract modifier along with the name of the method. And a class which has one or more abstract method is called as an abstract class. Since an abstract class is not concrete class, we cannot **instantiate that class.**(i.e it is not possible to create an instance of an abstract class). We say a class as **concrete or complete** only if that class has got implementation for all the methods in it. But since the abstract class doesn't provide implementation for one or more methods it is not complete and hence we cannot create object of that class. We need to override those abstract methods in the subclasses and we can invoke those methods by creating instances of the subclasses. To declare a abstract method, the syntax is

**abstract return-type methodname(list of parameters);**

We will implement the above example with the use of abstract class which will make clear about the concept of abstract classes.

**Eg:**

```
abstract class Shape
{
  int d1;
  int d2;
  public void setvalues(int dimen1, int dimen2 )
{
    d1 = dimen1;
    d2 = dimen2;
  }
 public abstract  void  area();
}
class Triangle extends Shape
 {
  int ar ;
  public void  area()
{
    ar = (d1 * d2 )/ 2;
    System.out.println("Area of triangle:"+ar);
  }
}
public class Inher
{
 public static void main(String args [ ])
{
//   Shape s = new Shape() ;    illegal as we cannot instantiate an abstract class
     Triangle t = new Triangle();
     t.setvalues(10,10);
     t.area();
 }
}
```

Since the method area in superclass Shape doesn't have any implementation, we have declared it as abstract. The class is declared abstract as it has got an abstract method. So an abstract class purpose is solely a design convenience. Abstract classes are made up of one or more abstract methods. These are methods that are declared but left bodiless. There are some limitations to using abstract:

- We cannot make abstract constructors
- Static and Private methods cannot be made abstract
- We cannot use abstract classes to instantiate objects
- Abstract methods of the abstract class must be defined in its subclass

## JAVA ACCESS MODIFIERS

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

1. Visible to the package, the default. No modifiers are needed.
2. Visible to the class only (private).
3. Visible to the world (public).
4. Visible to the package and all subclasses (protected).

**1. Default Access Modifier - No keyword:** Default access modifier means we do not explicitly declare an access modifier for a class, field, method etc. A variable or method declared without any access control modifier is available to any other class in the same package. The default modifier cannot be used for methods, fields in an interface.

**Example:** Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "1.5.1";
boolean processOrder()
 {  return true;}
```

**2. Private Access Modifier - private:** Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world. It is the highest degree of protection i.e. accessible only within own class. They cannot be inherited and so they are not accessible. A method declared as private acts as final method.

**3. Public Access Modifier - public:** A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However if the public class we are trying to access is in a different package, and then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses. It is the widest possible visibility and accessible everywhere.

**Example:** The following function uses public access control:

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

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

3. **Protected Access Modifier - protected:** Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected. Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it. It makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in

the other package. **NOTE:** Non-subclasses in other packages cannot access protected members.

**Access Control and Inheritance:**

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so there is no rule for them.

Modifiers are keywords that we add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

- **Java Access Modifiers**
- **Non Access Modifiers**

To use a modifier, we include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples:

```
public class className
{
  // ...
}
private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
public static void main(String[] arguments)
 {  // body of method}
```

**Non Access Modifiers:** Java provides a number of non-access modifiers to achieve much other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables
- The *abstract* modifier for creating abstract classes and methods
- The *synchronized* and *volatile* modifiers, which are used for threads

## STRING CLASS

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects. The Java platform provides the String class to create and manipulate strings. String class represents constant (immutable) strings.

1. **Creating Strings: String** class is defined in java.lang package. The Format of string constructors are

- String s = new String();
- String (char s [ ]);

The first constructor is used when we need to create an empty string. To create a String initialized by an array of characters the second constructor can be used. The simplest method and straightforward way to create a String object is to enclose the string literal in quotes and
assign the value to a String object. Creation of a String through assignment does not require the
use of the new operator, for example
String str = "abc";
String language = "Java";

Whenever it encounters a string literal in code, the compiler creates a String object with its value in this case, "Java!'. As with any other object, we can create String objects by using the new keyword and a constructor. The String class has eleven constructors that allow us to provide the initial value of the string using different sources, such as an array of characters:

```
public class StringDemo
{
  public static void main(String args[])
{
    char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
    String helloString = new
String(helloArray);
    System.out.println( helloString );
  }
}
```

This would produce following result:

```
hello
```

**Note:** The String class is immutable; so that once it is created a String object cannot be changed. Alternatively, String objects can be created through constructors. The following constructors are supported:

- public **String**() -Constructs a new String with the value **""** containing no characters. The value is not null.
- public **String**( String value )-Constructs a new String that contains the same sequence of characters as the specified String argument.

String s1 = new String();
String s2 = new String( "abc" ); // "abc" is first converted to a String and then passed as an String argument
String s3 = new String( s2 );

We can specify a subrange of a character array with the following constructor as

**String (Char chars [ ], int startindex, int numchars);**

2. **String Length:** Methods used to obtain information about an object are known as accessor methods. One accessor method that we can use with strings is the length() method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, len equals 17:

```
public class StringDemo
{  public static void main(String args[])
{     String palindrome = "Dot saw I was Tod";
    int len = palindrome.length();
    System.out.println( "String Length is : " + len );
  }
}
```

This would produce following result:

```
String Length is : 17
```

3. **String Concatenation:** String concatenation can be done with the help of + operator and using the concat() method.

**Eg:**

```
public class Democoncat
{
   public static void main(String args [ ] )
{
     String s1 = “10” ;
     String s2 = “I have” + s1 +”copies of my certificate” ;
      System.out.println(s2);
  }
}
```

Output: I have 10 copies of my certificate

Java Strings can be concatenated (joined) using the **+** and **+=** operators to create new Strings. String language = "Java"; String course = "Introduction to " + language; course += ", CS 101"; Every time an operation modifies a String object, a new read-only String object is created. In the above example, the += operator creates a new String object containing the characters "Introduction to Java, CS 101". The original memory containing "Introduction to Java" is unchanged and will be eventually collected by the Garbage Collector because the memory space is no longer referenced by a variable.

**Using concat()**

The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. We can also use the concat() method with string literals, as in:

**Eg:**

```
public class Democoncat
{
   public static void main(String args [ ] )
{
    String s1 = “concatenation”;
```

```
    String s2 = “string”;
    System.out.println(s2.concat(s1));
  }
}
```

The output is : string concatenation .

4. **Comparing Strings :**Java provides a variety of methods to compare String objects, including:
    1. public int **compareTo**( String str ) = Compares the current String object to str, and returns **0** only if the two strings contain the same sequence of characters (case sensitive). A negative value is returned if the current String is loour in the Unicode set than the String str. A positive value is returned if the current String is higher in the Unicode set than the String str.
    2. public boolean **equals**( Object obj ) = Compares the current String object with obj and returns true if obj is another String containing the same sequence of characters; false is returned otherwise.
    3. public boolean equalsIgnoreCase( String str ) = Performs a case-insensitive comparison of the current String object with str and returns true if str is another String containing the same sequence (ignoring case) of characters; false is returned otherwise.

Never use == to test equality of String contents.

**4. Useful String Methods:**

1. public char **charAt**( int index ) = Returns the character at the specified index. The index may range from 0 to length() - 1.
2. public String **toUpperCase**() = Converts the String to uppercase.
3. public String **toLowerCase**() = Converts the String to lowercase.
4. public String **trim**() = Removes white space from both ends of the String.
5. public static String valueOf( boolean b )
   public static String valueOf( char ch )
   public static String valueOf( int inum )
   public static String valueOf( long lnum )
   public static String valueOf( float fnum )
   public static String valueOf( double dnum ) = Creates the String representation of the argument.
6. boolean startsWith(String prefix)= Checks whether the string starts with the given string which is given as argument.
7. boolean endsWith(String prefix)= Checks whether the string ends with the given string which is given as argument.
8. boolean startsWith(String prefix, int index)=Checks whether the string starts with the given string which is given as argument from the specified index.
9. s2=s1.replace(‘x’,’y’)= Will replace character ‘x’ by character ‘y’ in the string s1 and assigns to string s2.
10. public String substring(int n)= Produces a substring starting from position “n” in the given string s1.
11. public String substring(int n, int m)=Produces a substring starting from position “n” to position “m” in the given string s1.

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

## STRINGBUFFER CLASS

It is the peer class of String class. While String class creates strings of fixed length, StringBuffer class will produce strings of variable length so that we can change the length and thereby content of strings if necessary.

Constructors used to create strings are:

1. StringBuffer()=This constructs a string buffer with no characters in it and an initial capacity of 16 characters.
2. StringBuffer(int length)= This constructs a string buffer with no characters in it and the specified initial capacity.
3. StringBuffer(String str)= This constructs a string buffer initialized to the contents of the specified string.

The useful methods of this class are:

1. StringBuffer(String s)= This method appends the specified string to this character sequence.
2. StringBuffer(StringBuffer SB)= This method appends the specified StringBuffer to this sequence.
3. int capacity()=This method returns the current capacity.
4. StringBuffer insert(int offset,String s)= This method inserts the string into this character sequence.
5. int length()=This method returns the length (character count).
6. void setLength(int newlength)= This method sets the length of the character sequence.

## JAVA UTILITIES

java.util package is the widely used package in Java with a wide assortment of classes and interfaces. The package java.util contains a number of useful classes and interfaces. Although the name of the package might imply that these are utility classes, they are really more important than that. In fact, Java depends directly on several of the classes in this package, and many programs will find these classes indispensable. The classes and interfaces in java.util include:

- The Hashtable classes for implementing hash tables or associative arrays.
- The Vector class, which supports variable-length arrays.
- The Enumeration interface for iterating through a collection of elements.
- The StringTokenizer class for parsing strings into distinct tokens separated by delimiter characters.
- The EventObject class and the EventListener interface, which form the basis of the new AWT event model.

- The Locale class which represents a particular locale for internationalization purposes.
- The Calendar and TimeZone classes in Java interpret the value of a Date object in the context of a particular calendar system.
- The ResourceBundle class
- Dictionary class in which key and value are objects and every key is associated with at most one value.
- Wrapper class
- ClassLoader class
- Random class is used to generate a stream of pseudorandom numbers.

**Example:**

1. **The Date Class**: Encapsulates date and time information like after(date), before(date),equals(date),toString() etc.
2. **The Calendar class:** Abstract class used to covert dates to fields like year, month, hour etc and it uses Gregorian calendar. Some methods to mention are getTime(), setTime(), setTimeZone(TimeZone) etc.
3. **Enumeration interface:** It is used to manipulate collection of objects with methods like hasMoreElements(), nextElement() etc.
4. **Vector Class:** Vector supports dynamic grow able arrays. Its constructors are vector(int), vector(int,int) and methods are addElement(object),capacity(),elementAt(),indexOf(Object), insertElementAt(object,int),isEmpty(),lastElement(),firstElement(),removeElement(object), removeElementAt(int),removeAllElements(), copyInto(array) etc.

   **Eg:**

```
//Vector class
import java.util.*;
class VectorClass
{public static void main(String args[])
{       Vector list=new Vector();
        for(int i=0;i<args.length;i++)
                list.addElement(args[i]);
        list.insertElementAt("Go",2);
        String array[]=new String(list.size);
        System.out.println("The vector elements are ");
        for(int i=0;i<list.size;i++)
                System.out.println(array[i]);
}
}
```

5. **The Stack class:** It is used to create and manipulate the basic operations of stack. The methods involved are empty(),peek(),pop(),push(object),search(object).

   **Eg:**

```
//Stack Program
import java.util.*;
public class StackPrg
```

```
{
public static void main(String args[])
{
        Stack s=new Stack();
        s.push("one");
        s.push("two");
        s.push("three");
        System.out.println(s.peek());
        while(!s.empty())
                System.out.println(s.pop());
}
}
```

6. Use of string tokenzier class is to separate group of strings based on a particular delimiter.

**Eg:**

```
import java.util.StringTokenizer;
public class MyTokens
{
   public static void main(String a[])
     {
      String msg = "This program gives sample code for String Tokenizer";
      StringTokenizer st = new StringTokenizer(msg," ");
      for (int i=0;i<st.countTokens();i++)
        System.out.println(st.nextToken());
      while(st.hasMoreTokens())
        System.out.println(st.nextToken());
     }
}
```

7. 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.

| No | Methods |
|---|---|
| 1 | **public static int binarySearch(Object[] a, Object key)**<br>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 -1. |
| 2 | **public static boolean equals(long[] a1, long[] a2)**<br>Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. |
| 3 | **public static void fill(int[] a, int val)**<br>Assigns the specified int value to each element of the specified array of int. |
| 4 | **public static void sort(Object[] a)** |

| | |
|---|---|
| | Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. |

8. **Date and Time Class:** Java provides the **Date** class available in **java.util** package; this class encapsulates the current date and time. The Date class supports two constructors. The first constructor initializes the object with the current date and time.

```
Date( )
```

The following constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970

```
Date(long millisec)
```

| No | Methods |
|---|---|
| 1 | **boolean after(Date date)**<br>Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false. |
| 2 | **boolean before(Date date)**<br>Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false. |
| 3 | **int compareTo(Date date)**<br>Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date. |
| 4 | **long getTime( )**<br>Returns the number of milliseconds that have elapsed since January 1, 1970. |
| 5 | **String toString( )**<br>Converts the invoking Date object into a string and returns the result. |

**Getting Current Date & Time:** This is very easy to get current date and time in Java. You can use a simple Date object with *toString()* method to print current date and time as follows:

```
import java.util.Date;
 class DateDemo
 {
   public static void main(String args[]) {
      // Instantiate a Date object
      Date date = new Date();
      // display time and date using toString()
      System.out.println(date.toString());
   }
}
```

This would produce following result:

```
Mon May 04 09:51:52 CDT 2009
```

**Date Comparison:** There are following three ways to compare two dates:

- We can use getTime( ) to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
- We can use the methods before( ), after( ), and equals( ). Because the 12th of the month comes before the 18th, for example, new Date(99, 2, 12).before(new Date (99, 2, 18)) returns true.
- We can use the compareTo( ) method, which is defined by the Comparable interface and implemented by Date.

**Date Formatting using SimpleDateFormat**

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting. For example:

```
import java.util.*;
import java.text.*;
class DateDemo
 {  public static void main(String args[])
{    Date dNow = new Date( );
     SimpleDateFormat ft =new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
     System.out.println("Current Date: " + ft.format(dNow));
  }
}
```

This would produce following result:

```
Sun 2004.07.18 at 04:14:09 PM PDT
```

**Simple DateFormat format codes:** To specify the time format use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

| Character | Description | Example |
|---|---|---|
| G | Era designator | AD |
| y | Year in four digits | 2001 |
| M | Month in year | July or 07 |
| d | Day in month | 10 |
| h | Hour in A.M./P.M. (1~12) | 12 |
| H | Hour in day (0~23) | 22 |
| m | Minute in hour | 30 |
| s | Second in minute | 55 |

| | | |
|---|---|---|
| S | Millisecond | 234 |
| E | Day in week | Tuesday |
| D | Day in year | 360 |
| F | Day of week in month | 2 (second Wed. in July) |
| w | Week in year | 40 |
| W | Week in month | 1 |
| a | A.M./P.M. marker | PM |
| k | Hour in day (1~24) | 24 |
| K | Hour in A.M./P.M. (0~11) | 10 |
| z | Time zone | Eastern Standard Time |
| ' | Escape for text | Delimiter |
| " | Single quote | ` |

**Eg:**

```
import java.util.Date;
 class DateDemo
{
  public static void main(String args[])
{
     // Instantiate a Date object
     Date date = new Date();
      // display time and date using toString()
     System.out.printf("%tc", "Current Time : ", date);
  }
}
```

This would produce following result:

```
Current Time: Mon May 04 09:51:52 CDT 2009
```

It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the index of the argument to be formatted. The index must immediately follow the %, and it must be terminated by a $. For example:

```
import java.util.Date;
class DateDemo
 {
  public static void main(String args[])
```

```
{
    // Instantiate a Date object
    Date date = new Date();
    // display time and date using toString()
    System.out.printf("%1$s %2$tB %2$td, %2$tY","Due date:", date);
  }
}
```

This would produce following result:

```
Due date: February 09, 2004
```

**GregorianCalendar Class:** GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. The **getInstance( )** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar. There are also several constructors for GregorianCalendar objects:

| No | Constructor with Description |
|---|---|
| 1 | **GregorianCalendar()**<br>Constructs a default GregorianCalendar using the current time in the default time zone with the default locale. |
| 2 | **GregorianCalendar(int year, int month, int date)**<br>Constructs a GregorianCalendar with the given date set in the default time zone with the default locale. |
| 3 | **GregorianCalendar(int year, int month, int date, int hour, int minute)**<br>Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale. |
| 4 | **GregorianCalendar(int year, int month, int date, int hour, int minute, int second)**<br>Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale. |

**Example:**

```
import java.util.*;
 class GregorianCalendarDemo
{
  public static void main(String args[])
{
    String months[] = {"Jan","Feb","Mar","Apr","May","Jun", "Jul", "Aug","Sep","Oct", "Nov",
"Dec"};
     int year;
    // Create a Gregorian calendar initialized with the current date and time in the default
```

```
locale and timezone.
    GregorianCalendar gcalendar = new GregorianCalendar();
    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(year = gcalendar.get(Calendar.YEAR));
    System.out.print("Time: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND));
      // Test if the current year is a leap year
    if(gcalendar.isLeapYear(year))
      System.out.println("The current year is a leap year");

    else
      System.out.println("The current year is not a leap year");
    }
}
```

This would produce following result:

```
Date: Apr 22 2009
Time: 11:25:27
The current year is not a leap year
```



## UNIT 3

**Packages - Access Protection - Importing Packages - Interfaces - Exception Handling - Throw and Throws - Thread - Synchronization - Messaging - Runnable Interface - Inter thread Communication - Deadlock - Suspending, Resuming and stopping threads - Multithreading.**

## INTERFACE

```
class A
{
   datatype var1;
   datatype var2;
   returntype method1(list of parameters)
```

```
        {
      body of method
        }
}
class B
{
   datatype var1;
   datatype var2;
   returntype method1(list of parameters)
        {
      body of method
    }
}
class C extends A,B
{
   datatype var1;
   datatype var2;
   returntype method1(list of parameters)
        {
      body of method
    }
}
```

The above declaration of **multiple inheritance** of classes is not supported by Java. Java supports the concept of multiple inheritance by extends keywords using a special type of class called as Interface. Interfaces are similar to classes but they lack instance variables and their methods are declared without any body. So interfaces can be called as **pure abstract classes**. Once defined any number of classes can implement an interface and one class can implement any number of interfaces. Any class that implements an interface must provide definition for the complete set of methods in the interface. Interfaces are designed to support dynamic method resolution at runtime. All variables are final variables here.

| CLASS | INTERFACE |
|---|---|
| Members can be constant or variables | Members should only be constants |
| Methods can be abstract or non-abstract | Methods should be only abstract |
| It can be instantiated to create objects | It cannot be used to create objects |
| It can use various access specifiers like public, private, protected etc. | It can use only public access specifiers. |

**Defining an Interface**

The general form of the interface is

```
interface interfacename
{

   type final-varname1 = value;
```

```
    type final-varname1 = value;
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    ……..
}
```

where **interfacename** is the name of the interface. The methods do not have any body and they just end with a semicolon since they are abstract methods. Variables in the interface are **final** and **static** which means that their value cannot be changed by the implementing class. These variables must be initialized with a constant value.

**Example:**

```
interface Area
{
    final static float PI=3.14F;
    float compute(float,float);
}
```

**Implementing interfaces**

To implement an interface include the implements clause in the class definition.

```
access class classname [extends superclass] [ implements interface [, interface…….]
{
     // class body
}
```

**Eg:** We will define a class that implements the above interface(Area)

```
class Rectangle implements Area
{
   public float computer(float x,float y)
         {       return(x*y);     }
}
class Circle implements Area
{
   public float computer(float x,float y)
         {       return(PI*x*x);           }
}
class Interface
{
public static void main(String args[])
{
Rectangle r=new Rectangle();
Circle c=new Circle()
Area a;
a=r;
System.out.println("The area of rectangle="+a.compute(10,20));
a=c;
System.out.println("The area of circle="+a.compute(10,0));
}
```

}

**Interfaces can be extended**

One interface can inherit another using the keyword extends. When a class implements an interface that inherits another interface, it must provide implementations for all the methods defined in the interface chain. The new sub interface will inherit all the members of super interface. **NOTE:** Interface cannot extends classes.

**Eg:**

```
interface interfacename extends interfacename1, interfacename2,.....
{
.....
}
```

```
interface I1
{
    int code=1001;
   String name="Fan";
}
interface I2
{
  void put();
}
interface Item extends I1,I2
{
----------
}
```



## PACKAGES

We already aware the ability to reuse the code is one of the OOP's advantages.

1. One way is to extend classes and implementing interface. But it is limited to reuse the classes within a program.
2. If we need to use classes in other programs, we have to go for packages (container of classes and interfaces)

When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

**Benefits**

1. Classes contained in packages of other programs can be easily used.
2. Two classes in two different packages can have the same name.
3. Provide a way to hide certain classes.
4. Separates 'design' from 'coding'. We can design classes and implement after deciding their relationship.
5. They allow organizing classes into units.

JAVA PACKAGES

Java API Packages

lang util io awt net applet

User-Defined Packages

**System Packages**

1. Use the fully qualified class name of the class Eg: java.awt.color
2. If we want to use class at many places of a program, use import statements.

**Import Statements**

In Java if a fully qualified name, which includes the package and the class name, is given then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class. For example following line would ask compiler to load all the classes available in directory java_installation/java/io:

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

Packages are containers for classes that are used to keep the class namespace compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported in to a new class definition.

**Creating our own package**

1. Declare the package at the beginning of the file as package packagename;
2. Define the class that is to be put inside the package (make it as public).
3. Create a subdirectory in the name of the package itself under the directory where the main source files are stored.
4. Compile the files in package as classname.java and it will produce the class file.
5. If our package contains more than one class, make a class as public and save the source file in that classname as classname.java. When a source file with many classes is compiled, Java creates many files with .class extensions.

**Eg:**

```
package package1;
public class ClassA
{
public void displayA()
{System.out.println("I am in the class A");}
}
package package2;
public class ClassB
{
protected int m=10;
public void displayB()
{
        System.out.println("I am in the class A");
```

```
        System.out.println("m="+m);
}
}
import package1.*;
import package2.*;
class PackageMain
{
public static void main(String args[])
{
ClassA a=new ClassA();
ClassB b=new ClassB();
a.displayA();
b.displayB();
}
}
```

**Access-Control**

| | Private | Protected | Public |
|---|---|---|---|
| Same Class | √ | √ | √ |
| Subclass in same package | x | √ | √ |
| Other classes in same package | x | √ | √ |
| Subclass in other package | x | √ | √ |
| Non-subclass in other packages | x | x | √ |

**Eg:**

```
package p1;
public class X
{//available outside
}
class Y
{//Not available outside
}
```

## EXCEPTION HANDLING

Programs with mistakes will produce unexpected results. Errors are the wrongs that make a program to go wrong. Detection and correction of wrong are necessary. Java usually avoids dangerous errors like pointers. Apart from extensive error checking, errors may occur. There are two types of errors as Compile-time errors and Run-time errors.

**Compile-time errors:** Syntax errors are detected by Java Compiler-.class files is not created. Compiler tells the error along with line number for fixing errors.

**Eg:**

- Missing semicolons
- Missing brackets
- Misspelling of keywords

- Missing double quotes
- Use of undefined variables
- Incompatible datatypes
- Bad reference to objects
- Use of = in place of == etc.

**Run-time errors:** After creating .class file, program may not properly run-they are called as run-time errors.

**Eg:**

- Divide by zero
- Out of bounds of array
- Store a value of array of an incompatible class
- Trying to cast an instance of class to its subclass
- Passing a parameter that is not a valid range
- Illegally change the state of a thread
- Negative size of an array
- Null object reference
- Converting invalid string to number
- Accessing a character that is out of bounds of a string etc.

**Exception**: An exception is an abnormal condition that arises in a code sequence at runtime (or exception is a runtime error). A Java exception is an object that describes an exceptional condition that has occurred in a piece of code. When an exceptional condition arises, an object representing the exception is created and it is thrown in the method that caused the error. The method can handle the exception or pass it. Exceptions can be generated by Java run-time system or they can be manually generated by our code. If we want the program to execute with the remaining code, we should try to catch the exception object and display a message for taking actions which is called as exception handling. To understand how exception handling works in Java, we need to understand the three categories of exceptions:

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in our code because we can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

**Basic Steps**

1. Find the problem (hit)
2. Inform about error (throw)
3. Receive the error (catch)
4. Take corrective measures (handle)

**Common Java Exceptions**

- ArithmeticException
- ArrayIndexOutOfBoundsException
- ArrayStoreException
- FileNotFoundException
- IOException
- NullPointerException
- NumberFormatException
- OutOfMemoryException
- SecurityException
- StackOverFlowException
- StringIndexOutOfBoundsException

**Keywords in Exception Handling**

The basic keywords used in exception handling are **try, catch, throw, throws, finally.** The statements that are to be monitored for any errors must be put inside the **try** block. If an exception occurs in the **try** block it is thrown. The code can **catch** that exception and handle it. To manually throw an exception use the keyword **throw.** All the exceptions that occur in a method have to be specified by the word **throws.** Any code that must be executed before a method returns is put in **finally** block.

**Syntax:**

```
try
{
    Block of code to be monitored for errors
}
catch(ExceptionType1 exobj)
{
      //exception handler for ExceptionType1
 }
catch(ExceptionType2 exobj)
{
        //exception handler for ExceptionType2
}
finally
{
 //block of code to be executed before try block ends
}
```

where Exceptiontype is the type of exception that has occurred.

try block should be followed by at least one catch statement else compilation error occurs. If the catch parameter matches with the type of exception object, then the exception is caught; if not, default exception handler will terminate the program. A program can be nested try and multiple catch statements.

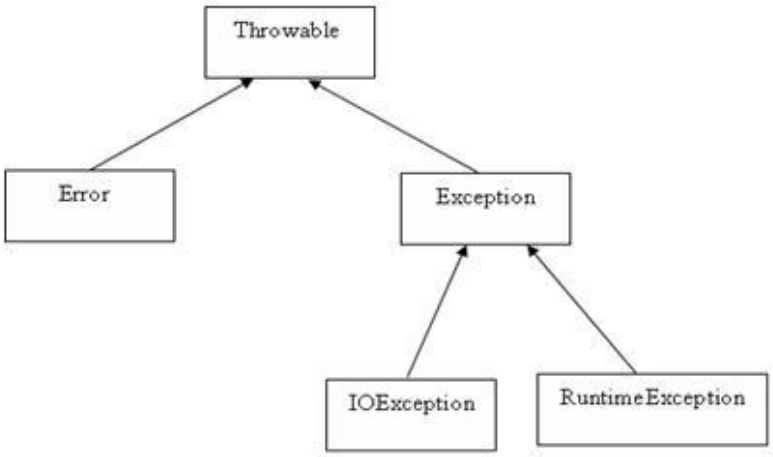
Exception handling gives two advantages.

1. It allows we to fix errors
2. It prevents the program from automatically terminating

**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 not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which 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.




**NOTE: public String getMessage()** will return a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

**Uncaught exceptions:** Consider the following program

```
public class Ex
{
   public static void main(String args[ ])
{
        int d = 0;
        int a = 50 / d ;
   }
}
```

We are making an attempt to divide a value by zero which results in an exception. Java runtime system detects this error and constructs a new exception object and throws the object. This causes the execution of the program to terminate. Since the exception is not caught here it will be handled by the default handler and the default handler just displays a string describing the exception and terminates the program. The output of the above program will be

java.lang.ArithmeticException : / by zero at Ex.main(Ex.java.4)

The system specifies the class name and the method name where the exception occurs along with the line number. It keeps track of path of the exception. It specifies the type of the exception (ArithmeticException).

**Catching Exceptions**

A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}
catch(ExceptionName e1)
{
  //Catch block
}
```

A catch statement involves declaring the type of exception we are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follow the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

**Eg:**

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest
{
  public static void main(String args[])
{
    try
   {
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }
  catch(ArrayIndexOutOfBoundsException e)
   {
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

**Eg. for exception programs:**

class Except1

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

  {             //Declare an array of strings
                String Box [] = {"Book", "Pen", "Pencil"};
                for(int i=0;i<4;i++)
                {
                        try
                        {System.out.println(Box[i]);
                        }
                        catch(ArrayIndexOutOfBoundsException e)
                        {System.out.println("Subscript Problem " + e);
                        }
                        finally
                        {System.out.println("This is always printed.");
                        }
                }
  }
}
```

---

```
class Except2
{
   public static void main (String args[]) {

    /* Now let's say hello */
    try {
      System.out.println("Hello " + args[0]);
    }
    catch (SecurityException e)
          {     System.out.println("Hello whoever you are");
          }
  }
}
```

---

```
public class Except3
{

  public static void main(String args[]){
    int a[] = new int[2];
    try{
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
```

```
    }
    finally{
      a[0] = 6;
      System.out.println("First element value: " +a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

---

```
class Except4
{
public static void main(String args[])

  {
     try
      {  int i = 10/0;
     }
catch(ArithmeticException ex)
     {
       System.out.println("Inside 1st catch Block");
     }
finally
 {
       System.out.println("Inside 1st finally block");
     }

     try
    {
       int i = 10/10;
     } catch(ArithmeticException ex){
       System.out.println("Inside 2nd catch Block");
     } finally {
       System.out.println("Inside 2nd finally block");
     }
  }
}
```

---

```
class Except5
{
public static void main(String args[])

  {          //Declare an array of strings
             String Box [] = {"Book", "Pen", "Pencil"};
             for(int i=0;i<4;i++)
             {
```

```
                    try
                    {System.out.println(Box[i]);}
                    catch(ArrayIndexOutOfBoundsException e)
                    {System.out.println("Subscript Problem " + e);
                    }
                    finally
                    {System.out.println("This is always printed.");
                    }
                }
  }
}
```

---

```
import java.io.*;
import java.util.*;
public class Exception
{
void arith()
{
int a,b;
try
{
        System.out.println("1. Inside Arithmetic");
        a=0;
        b=10/a;
        System.out.println("a="+a+" b="+b);
}
catch(ArithmeticException e)
{System.out.println("\t"+"Divide by zero is an arithmetic exception\n"+"\t"+e);}
}
void array()
{
try
{
System.out.println("2. Inside Array");
int a[]={1,2,3,4,5};
 a[10]=78;
}
catch(ArrayIndexOutOfBoundsException e)
{System.out.println("\t"+"Wrong index in the given array"+"\n"+"\t"+e);}
}
void num()
{
Scanner s=new Scanner(System.in);
int x[]=new int[5];
System.out.println("Enter 5 elements:");
try
```

```
{
        for (int i=0;i<5;i++)
         x[i]=Integer.parseInt(s.nextLine());
}
catch(NumberFormatException e)
{System.out.println("\t"+"Wrong input"+"\n"+"\t"+e);}
}
public static void main(String args[]) throws IOException
{
Exception obj=new Exception();
obj.arith();
obj.array();
obj.num();
}
}
----------------------------------------------------------------------------------------------------class
CommandLine
{
public static void main(String args[])
{
int invalid=0,number,count=0;
for(int i=0;i<args.length;i++)
{
    try
    {number=Integer.parseInt(args[i]);}
    catch(NumberFormatException e)
    {invalid++;
    System.out.println("Invalid arg="+args[i]);
    continue;
    }
    count++;
}
System.out.println("Valid="+count);
System.out.println("Invalid="+invalid);
}
}
```

**The finally Keyword**

When exceptions occur, execution in a method takes a non-linear path. It is also possible for execution to cause the method to return prematurely. finally keyword is used to avoid this. finally will execute whether or not an exception is thrown. It can be useful for closing file handles and free up any other resources. Finally clause is optional. But each try statement must have at least one catch or finally clause. The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes (guarantee of execution), whether or not an exception has occurred. Using a finally block allows we to run any cleanup-type statements that we want to execute, no matter what

happens in the protected code. A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
  //Protected code
}
catch(ExceptionType1 e1)
{
  //Catch block
}
catch(ExceptionType2 e2)
{
  //Catch block
}
catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

**Eg:**

```
public class ExcepTest
{
  public static void main(String args[])
{
    int a[] = new int[2];
    try
  { System.out.println("Access element three :" + a[3]);}
    catch(ArrayIndexOutOfBoundsException e)
   {System.out.println("Exception thrown  :" + e);}
    finally
    {
      a[0] = 6;
      System.out.println("First element value: " +a[0]);
      System.out.println("The finally statement is
executed");
    }
  }
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

**NOTE:**

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

**Nested try statements**

A try statement can be inside another try block. If an inner try block doesn't have a catch handler then the next try block's catch handlers are inspected for a match. This is continued until one of the catch statement succeeds or until all the try blocks have been exhausted. If no catch statement matches the Java run-time system will handle the exception.

```
class Nesttry
{
  public static void main(String args[ ])
{
   try
        {
                int a = 0;
                int b = 42 / a;
                System.out.println("a " +a);
                try
                        {
                         int c [ ] = { 1};
                        c[ 42] = 99 ;
                }
                catch(ArrayIndexOutOfBoundsException e)
                {  System.out.println("Index out of range");  }
                catch(ArithmeticException e)
                {  System.out.println("Divide by zero");  }
        }
}
}
```

In the outer try block we are trying to assign a value to an array where index or subscript is out of range. The exception that occurs is ArrayIndexOutOfBoundsException which is handled and the outer block has ArithmeticException.

**Multiple catch Blocks**

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
//Protected code
}
catch(ExceptionType1 e1)
```

```
{
//Catch block
}
catch(ExceptionType2 e2)
{
//Catch block
}
catch(ExceptionType3 e3)
{
//Catch block
}
```

The previous statements demonstrate three catch blocks, but we can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

**Eg:**

```
class MultipleCatch
{
public static void main(String args[])
{
int a[]={5,10};
int b=5;
try
{int x=a[2]/b-a[1];
catch(ArithmeticException e)
{System.out.println("Division by zero");}
catch(ArrayIndexOutOfBoundsException e)
{System.out.println("Array Index error");}
catch(ArrayStoreException e)
{System.out.println("Wrong data type");}
int y=a[1]/a[0];
System.out.println(y);
}
}
```



**Declaring own Exception**

We can create our own exceptions in Java. All exceptions must be a child of Throwable.

**Syntax:**

**throw new throwable subclass;**

**Eg 1:**

```
class MyException extends Throwable
{
```

```
int accno[] = {1001,1002,1003,1004,1005};
String name[] = {"Hari","Siva","Bhanu","Rama","Chandu"};
double bal[] = {2500,3500,1500,1000,6000};
MyException(String S)
{super(S);}
public static void main(String args[])
{
        MyException me = new MyException("");
        System.out.println("AccNo \t Name \t Balance ");
        for(int i=0;i<5;i++)
        {
                try
                {
                System.out.println(me.accno[i]+ "\t" + me.name[i] + "\t" +me.bal[i] );
                if( me.bal[i] < 2000 )
                        throw new MyException("InsufficientBalance");
                }
                catch(MyException e)
                {
                System.out.println(e);
                continue;
                }
        }
}
}
```

---

**Eg 2:**

```
import java.lang.Throwable;
class MyException1 extends Throwable
{
        MyException1(String message)
        {super(message);}
}

class MainPrg
{public static void main(String args[])
{
        int x=5,y=100;
        try
        { float z=(float)x/(float)y;
         if (z<0.01)
                throw new MyException1("Too small number");
        }
        catch(MyException1 e)
        {
                System.out.println("Caught the exception");
```

```
                System.out.println(e.getMessage());
        }
        finally
        { System.out.println("End of program");}
}
}
```

**The throws/throw Keywords**

Throws keyword helps to print user-defined exception. There could be situations where no exception handling is prevalent to this method. So, it is to be informed earlier that certain types of exceptions may occur in future. If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature. We can throw an exception, either a newly instantiated one or an exception that we just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

**Syntax:**

```
returntype method(arguments) throws ExceptionList
{  ----- }
```

**Eg:**

```
class Example
{
static void div() throws ArithmeticException
{
int x=22,y=0,z;
z=x/y;
}
public static void main(String args[])
{
        try
        {div();}
        catch(ArithmeticException e)
        {System.out.println(e);}
}
}
```

## THREADS

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes. A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist of its own; it must be a part of a process.

A process remains running until all of the non-daemon threads are done executing. Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. Ability of a language to support threads is called concurrency. Threads in Java are subprograms of a main program and share the

same memory space called as Light Weight Process (LWP). Threads actually do not run in parallel in single processor – instead only the flow of execution is shared which increases the speed of program.

**Eg:**

```
class ABC
{
……..
……..
……..
}
```

| MULTITHREADING | MULTITASKING |
|---|---|
| Method of dividing a program into threads | Operating System concept of doing multiple tasks simultaneously |
| Supports the execution of multiple parts of a single program | Supports the execution of multiple programs |
| Processor has to switch between different parts of a same program | Processor has to switch between different programs |
| Highly efficient | Less efficient than multithreading |
| Smallest unit of multithreading is thread | Smallest unit of multitasking is program |
| Helps in developing efficient programs | Helps in developing efficient operating system |
| Cost-effective in context switching | Expensive |

**Java Thread model**

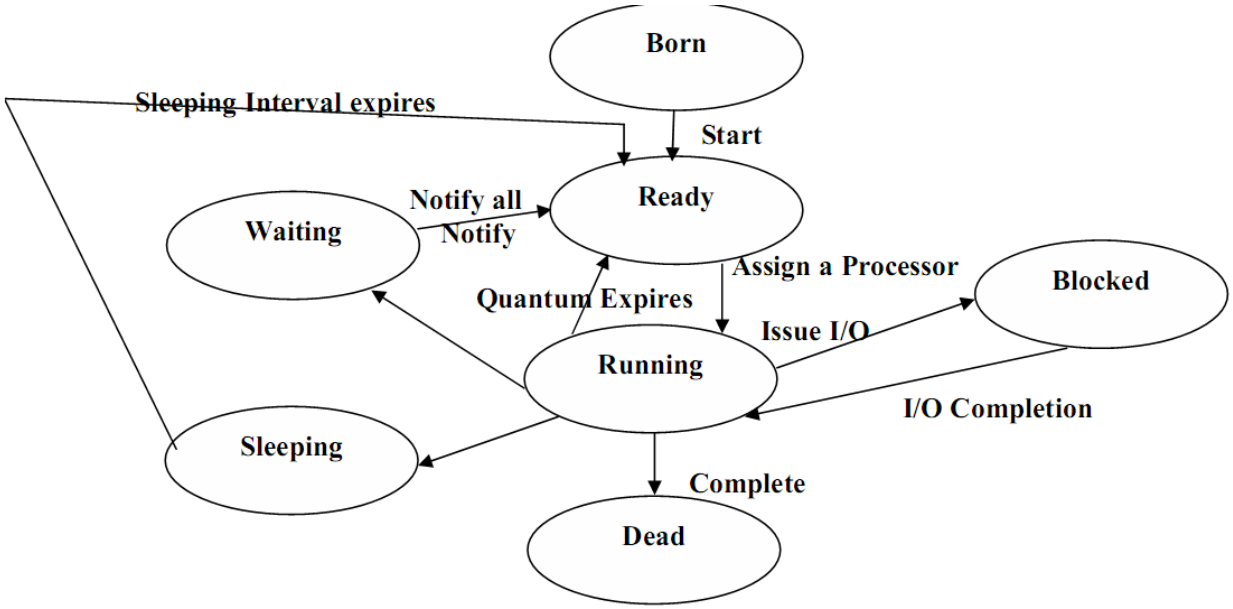
In single-threaded systems, when a thread blocks because it is waiting for some resource, the entire program stops running, this wastes CPU time. This drawback is overcome by multi-threading. Here one thread can pause without stopping other parts of our program. The idle time can be utilized somewhere else. So when a thread blocks a Java program, only the single thread that is blocked pauses. All other threads continue to run.

**Uses of Threads**

- Threads are used in designing serverside programs to handle multiple clients at a time.
- Threads are used in games and animations.
- We can reduce the idle time of processor.
- Performance of processor is improved.
- Reduces interferences between execution and user interface.

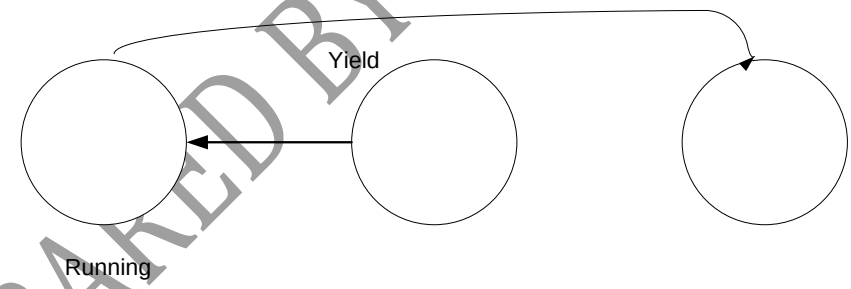
**Thread states**

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.

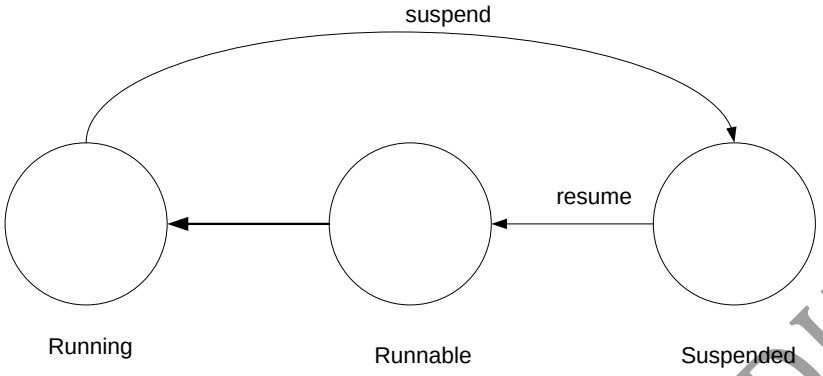



Above mentioned stages are explained here:

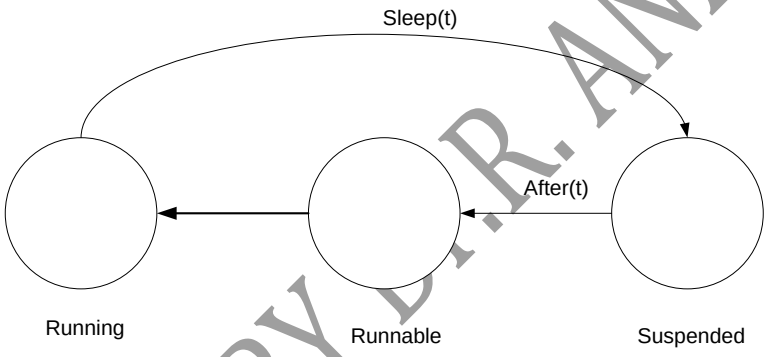
- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** It refers to ready for execution and waiting for its processor turn ie joined in the queue. If all are of equal priority and execution in round-robin fashion by time slice. If we want a thread to relinquish control to other, yield() method will be useful.



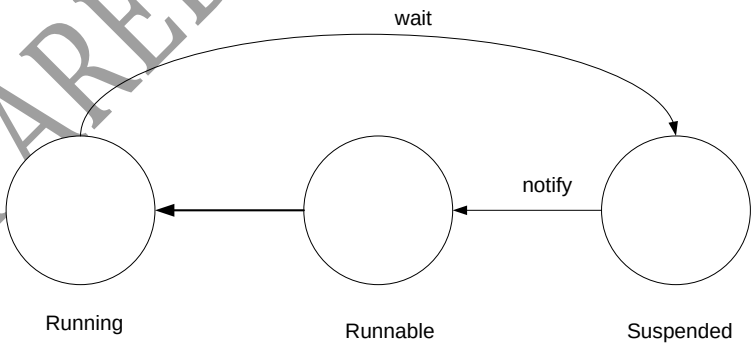

- **Running:** A thread in this state is considered to be executing its task. It may give the control to others only
  a. if it is suspended by suspend()-this is useful if we want to suspend a thread for some time, but not killing it.




b. if it has been made to sleep by sleep(time) method where time is in ms. The thread will be kept out the queue during this time and after time elapses, the thread reenters the queue.




c. it has been told to wait until some event occurs with the help of wait() method. Again it can be made to run by notify() event.






- **Waiting:** Sometimes a thread transitions to the waiting state [preventing from entering into running by suspend(),wait() or sleep()] 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 these state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates. A thread may end its execution normally or make to die by stop() method refers to premature death.

**Creating Threads**

Threads are implemented in the form of objects that contain a method run()-Heart and soul of the thread which makes the entire body of a thread.

```
public void run()
{……..
……..
}
```

A thread can be created by two ways as,

1. By creating a thread class – define a class that extends Thread class and override its run() method.
2. By converting a class to a thread – define a class that implements Runnable interface with a method run().

**1. Extending the Thread class**

The steps are

a. Declare a class by extending the Thread class.
```
class MyClass extends Thread
```
b. Define and implement the run() method.
```
public void run()
{
}
```
c. Create a thread object and activate by start() method.
```
MyClass obj=new MyClass();
obj.start();
```

To create a thread, we can use the following forms of constructors:

1. Thread t=new Thread();
2. Thread t=new Thread(obj);
3. Thread t=new Thread(obj,"thread name");

**2. Implementing 'Runnable' Interface**

a. Declare a class by implementing the Runnable interface.
```
class MyClass implements Runnable
```
b. Define and implement run() method.
```
public void run()
{   }
```
c. Create an object (thread) for the above defined class.
```
MyClass obj=new MyClass();
```
d. Call using start() method.
```
obj.start();
```

**Thread API Summary**

1. **currentThread() =**This method returns the thread object which is the currently running thread
2. **yield()=** It causes the runtime to context switch from the current running thread to the next available runnable thread. This is the only way to ensure that threads at loour priority do not get starved.

3. **sleep(int n) =**This method causes the run-time to put the current thread to sleep for some milliseconds. After n millisecond is expired, this thread will eligible to run again.
4. **start() =**This method is to create a system thread and start its running. run() method on this thread's target will be called in the new thread context. The start method cannot be called more than once on a given thread object
5. **run() =** This is the body of the running thread. It is called by the start method after the thread has been initialized. Whenever the run method returns the current thread will stop.
6. **stop() =**This causes the thread to stop immediately. This is an abrupt way to end a thread.
7. **suspend()=**It stops running of thread without destroying the underlying system thread. If a thread is suspended, we can call resume() on the same thread to cause it to start running again.
8. **resume()=**It is used to revive a suspended thread.
9. **setPriority(int p)=**Sets the thread's priority to the value passed in.
10. **getPriority()=**Returns the thread's current priority
11. **setName(String name) =**Used to give a name for the thread
12. **getName()=**Returns the current string value of the thread's name.
13. **isAlive()=**To see if a thread is still running
14. **join()=**Wait for a thread to terminate
15. **wait()-**Wait till the object is released

**Eg:**

```
//Creating and running a Thread
class MyThread extends Thread
{
public void run ()
{ for (int i = 0;i<100;i++)
     System.out.print (i + "\t");
}
}
class TDemo
{ public static void main(String args[])
{
MyThread obj = new MyThread ();
Thread t = new Thread (obj);
t.start ();
}
}
```

---

```
// Illustration about threads
public class Threads
{
public static void main(String args[])
{
A a=new A(); B b=new B();C c=new C();
```

```
a.start(); b.start();c.start();
}
}
class A extends Thread
{
public void run()
{
System.out.println("A Started\n");
for (int i=1;i<=5;i++)
{
  if (i==1) yield();
  System.out.println("From thread A:i="+i);
}
System.out.println("Exited from A");
}
}
class B extends Thread
{
public void run()
{
System.out.println("B Started\n");
for (int i=1;i<=5;i++)
{
  if (i==3) stop();
  System.out.println("From thread B:i="+i);
}
System.out.println("Exited from B");
}
}
class C extends Thread
{
public void run()
{
System.out.println("C Started\n");
for (int i=1;i<=5;i++)
  System.out.println("From thread C:i="+i);
System.out.println("Exited from C");
}
}
```

---

```
//Runnable interface
class X implements Runnable
{ public void run()
{
for(int i=1;i<=5;i++)
        System.out.println(“Thread X=”+i);
```

```
System.out.println("End of thread");
}
}
class RunnableClass
{public static void main(String args[])
{X obj=new X();
Thread tx=new Thread(obj);
tx.start();
}
}
```

---

```
//Usage of more than 1 thread
class Theatre extends Thread
{ String str;
Theatre (String str)
{ this.str = str;
}
public void run()
{ for (int i = 1; i <= 10 ; i++)
{ System.out.println (str + " : " + i);
try
{ Thread.sleep (2000);
}
catch (InterruptedException ie) { ie.printStackTrace (); }
}
}
}
class TDemo1
{ public static void main(String args[])
{
Theatre obj1 = new Theatre ("Reserve Ticket");
Theatre obj2 = new Theatre ("Show Seats");
Thread t1 = new Thread (obj1);
Thread t2 = new Thread (obj2);
t1.start ();
t2.start ();
}
```

---

```
//Multiple Threads acting on single object
class Reserve implements Runnable
{ int available = 3;
int wanted;
Reserve (int i)
{ wanted = i;
}
public void run()
```

```
{ synchronized (this)
{ System.out.println ("Number of berths available: " + available);
if ( available >= wanted)
{ String name = Thread.currentThread ().getName ();
System.out.println (wanted + " berths alloted to: " + name);
try
{ available = available - wanted;
Thread.sleep (2000); // wait for priniting the ticket
}
catch (InterruptedException ie)
{ ie.printStackTrace (); }
}
else
{ System.out.println ("Sorry, no required berths available");
}
}
}
}
class Safe
{ public static void main(String args[])
{ Reserve obj = new Reserve (2);
Thread t1 =new Thread (obj);
Thread t2 = new Thread (obj);
t1.setName ("First Person");
t2.setName ("Second Person");
t2.start ();
t1.start ();
}
}
```

---

**The Main thread**

When a Java program starts one thread begins execution automatically. It is called the main thread of the program. It is the last thread to finish execution (shutdown actions will take place here). It is the thread from which other child can be spawned.

**Eg:**

```
// Knowing about main thread
class Current
{public static void main(String args[])
{System.out.println ("This is first statement");
Thread t = Thread.currentThread ();
System.out.println ("Current Thread: " + t);
System.out.println ("Its name: " + t.getName ());
System.out.println ("Its priority:" + t.getPriority ());
}
```

```
}
```

```
class CurrentThreadDemo
{
   public static void main(String args [ ])
{
      Thread t = Thread.currentThread();
       System.out.println("Current thread :" +t);
        t.setName("My thread");
        System.out.println("After name Change: " + t);
         try{
             for (int i = 5 ; i > 0 ; i-- )
           {
              System.out.println(i);
               Thread.sleep(1000);
            }
           }
           catch(InterruptedException e)
           {System.out.println("Main thread interrupted");}
 }
}
```

A reference to the current thread can be obtained by the method currentThread(). The default name for the thread is main and after name change it becomes My Thread.

**Thread Priorities**

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java 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. Thread priority is used to decide when to switch between one running thread and another. This is called as **Context Switch.** The following rules determine when a context switch takes place.

1. A thread can voluntarily give-up control
2. A thread can be pre-empted by a higher-priority thread

To set a thread's priority use the method **final void setPriority(int level) ;** where level specifies the value for priority. It must be in the range of MIN_PRIORITY to MAX_PRIORITY. The corresponding values are 1 and 10 respectively. And the default priority is 5 which can be specified as NORM_PRIORITY.

**Eg:**

```
// Use of priority in threads
```

```
class A extends Thread
{
public void run()
{
System.out.println("Thread A started");
for (int i=1;i<=4;i++)
  System.out.println("From Thread A:"+i);
System.out.println("Exit from A");
}
}
class B extends Thread
{
public void run()
{
System.out.println("Thread B started");
for (int i=1;i<=4;i++)
  System.out.println("From Thread B:"+i);
System.out.println("Exit from B");
}
}
class C extends Thread
{
public void run()
{
System.out.println("Thread C started");
for (int i=1;i<=4;i++)
  System.out.println("From Thread C:"+i);
System.out.println("Exit from C");
}
}
public class ThreadPriority
{
public static void main(String args[])
{
A a=new A(); B b=new B(); C c= new C();
c.setPriority(Thread.MAX_PRIORITY);
b.setPriority(2);
a.setPriority(Thread.MIN_PRIORITY+3);
a.start();
b.start();
c.start();
}
}
```

**Thread Synchronization**

When two or more threads need to access a shared resource, they need some way to ensure that only one thread access the resource at a time. The process by which this is achieved is called as **synchronization.** Synchronization uses the concept of monitor (semaphore). Only one thread can own a monitor at a time. All other threads must wait for the monitor until the thread leaves the monitor. In this program nothing stops the 2 threads from calling the same method on the same object at the same time. This is known as **race condition** as all the threads are racing to complete the method. To avoid this precede the method definition call with the keyword **synchronized.** When a thread is acting on an object preventing other threads from acting on the same object is called Thread Synchronization or Thread Safe. The object on which the threads are synchronized is called 'synchronized object'.

The Object on which the Threads are synchronized is called synchronized object or Mutex (Mutually Exclusive Lock).Synchronized object is like a locked object, locked on a thread. It is like a room with only one door. A person has entered the room and locked form it from behind. The second person who wants to enter the room should wait till the first person comes out. In this way, a thread also locks the object after entering it. Then the next thread cannot enter it till the first thread comes out. This means the object is locked mutually on threads. So, this object is called 'mutex'. Synchronization of threads can be made by two ways:

1) To synchronize an entire method code, we can use synchronized keyword before method name.
   **synchronized void method()**
   **{**
   **……**
   **……**
   **}**
2) We can synchronize a block of statements. Though creating synchronized methods is an easy way of achieving synchronization it will not work in all cases. When we are using a class which is created by someone, we may not be able to access the source code and we can't add synchronized keyword to the methods. The solution to this is we can put calls to the methods that are defined by the class inside a synchronized block.
   **synchronized(object)**
   **{……**
   **……}**

**Eg 1:**

```
//Thread synchronization- Two threads acting on same object
//Multiple Threads acting on single object
class Reserve implements Runnable
{  int available = 1;
  int wanted;
  Reserve (int i)
  {  wanted = i;  }
  public void run()
  {  synchronized (this)
    {      System.out.println ("Number of berths available: " + available);
```

```
        if ( available >= wanted)
                {       String name = Thread.currentThread ().getName ();
                        System.out.println (wanted + " berths alloted to: " + name);
                        try
                        {  Thread.sleep (2000); // wait for priniting the ticket
                        available = available - wanted;
                        }
                        catch (InterruptedException ie)
                        {  ie.printStackTrace ();   }
                }
                else
                        {   System.out.println ("Sorry, no berths available");  }
                }
        }
       }
class Safe
{  public static void main(String args[])
 {  Reserve obj = new Reserve (1);
   Thread t1 =new Thread (obj);
   Thread t2 = new Thread (obj);
   t1.setName ("First Person");
   t2.setName ("Second Person");
   t1.start ();
   t2.start ();
 }
}
```

**Eg 2:**

```
class A extends Thread
{
synchronized public void run()
{
try
{while(true)
{
sleep(10);
System.out.println("good morning");
}
}
catch(Exception e)
{      }
}
}
class B extends Thread
{
synchronized public void run()
```

```
{
try
{
while(true)
{
sleep(20);
System.out.println("hello");
}
}
catch(Exception e)
{       }
}
}
class C extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
sleep(30);
System.out.println("welcome");
}
}
catch(Exception e)
{       }
}
}
class ThreadDemo
{
public static void main(String args[])
{
A t1=new A();B t2=new B();C t3=new C();
t1.start();
t2.start();
t3.start();
}
}
```

**Interthread Communication**

It refers to exchange of messages between two threads. It can be implemented by three methods. One thread output may be send as input to other thread. For example, a consumer thread is waiting for a producer to produce the data (or some goods). When the producer thread completes production of data, then the consumer thread should take that data and use it. In producer class, we take a StringBuffer object to store data. In this case, we take some numbers from 1 to 5. These numbers are added to StringBuffer object. Until

producer completes placing the data into StringBuffer the consumer has to wait. Producer sends a notification immediately after the data production is over.

1. notify()- Resumes the first thread that went to run.
   **final void notify()**
2. notifyall()- Resumes all threads that are ready for execution starts as per priority.
   **final void notifyAll()**
3. wait()- All calling threads are made to sleep.
   **final void wait()**

**NOTE:** All three methods throw InterruptedException.

**Eg:**

```
class Producer implements Runnable
{
        Thread1 t;
        Producer(Thread1 t)
        {
                this.t=t;
                new Thread(this,"Producer").start();
        }
        public void run()
        {
                int i=0;
                while (true)
                {
                        t.put(i++);
                }
        }
}
class Consumer implements Runnable
{
   Thread1 t;
        Consumer(Thread1 t)
        {
                this.t=t;
                new Thread(this,"Consumer").start();
        }
public void run()
        {
                int i=0;
                while (true)
                {
                        t.get();
                }
        }
}
class  ProducerConsumer
{
```

```
        public static void main(String[] args)
        {
                Thread1 t=new Thread1();
                System.out.println("Press Control+c to exit");
                new Producer(t);
                new Consumer(t);

        }
}
```

**Output:**

```
Press Control+C to exit
Put:0
Get:0
Put:1
Get:1
```

**Thread Deadlock**

Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects. Suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread X tries to call any synchronized method in Y it will block as expected. If the thread in Y tries to call any synchronized method in X then the thread waits forever. Deadlock is difficult to debug for 2 reasons.

1. It occurs when the two threads time-slice.
2. It involves more than two threads and two synchronized objects.

**Eg:**

A thread can be employed to execute one task at a time. Suppose there are 3 tasks to be executed. We can create a thread and pass the 3 tasks one by one to the thread. For this purpose, we can write all these tasks separately in separate methods; task1(), task2(), task3(). Then these methods should be called from run() method, one by one. Remember, a thread executes only the code inside the run() method. It can never execute other methods unless they are called from run().

**Eg:**

```
class MyThread implements Runnable
{
        public void run()
        {
                //executes tasks one by one by calling the methods.
                task1();
                task2();
                task3();
        }
        void task1()
        {System.out.println("this is task1");}
        void task2()
        {System.out.println("this is task2");}
```

```
        void task3()
        {System.out.println("this is task3");}
}



class Sin
{
        public static void main(String args[])
{
                MyThread obj=new MyThread();
                Thread t1=new Thread(obj);
                t1.start();
        }
}
```

**Multi Tasking using Threads**

In multi tasking, several tasks are executed at a time. For this purpose, we need more than one thread. For example, to perform 2 tasks, we can take 2 threads and attach them to the 2 tasks. Then those tasks are simultaneously executed by the two threads. Using more than one thread is called 'multi threading'.

**Program : Write a program to create more than one thread.**

```
//using more than one thread is called Multi Threading
class Theatre extends Thread
{  String str;
  Theatre (String str)
  {  this.str = str;
  }
public void run()
  {  for (int i = 1; i <= 10 ; i++)
    {  System.out.println (str + " : " + i);
      try
      {  Thread.sleep (2000);
}
      catch (InterruptedException ie)   {    ie.printStackTrace ();    }
    }
  }
}
class TDemo1
{  public static void main(String args[])
  {  Theatre obj1 = new Theatre ("Cut Ticket");
    Theatre obj2 = new Theatre ("Show Chair");
    Thread t1 = new Thread (obj1);
    Thread t2 = new Thread (obj2);
    t1.start ();
    t2.start ();
```

```
 }
}
```

In the preceding example, we have used 2 threads on the 2 objects of TDemo1 class. First we have taken a String variable str in Theatre class. Then we passed two strings- cut ticket and show chair into that variable from TDemo1 class. When t1. start () is executed, it starts execution run () method code showing cut ticket. Note that in run () method, we used: Thread. sleep (2000) is a static method in Thread class, which is used to suspend execution of a thread for some specified milliseconds. Since this method can throw InterruptedException, we caught it in catch block. When Thread t1 is suspended immediately t2. start () will make the thread t2 to execute.

**Multiple Threads Acting on Single Object**

First let us see why 2 threads should share same object (same run()method). We write an object to represent one task. If there is a different task, we take another object. When two people (threads) want to perform same task then they need same object (run () method) to be executed each time. Take the case of railway reservation. Every day several people want reservation of a berth for them. The procedure to reserve the berth is same for all the people. So we need some object with same run () method to be executed repeatedly for all the people (threads). Let us think that only one berth is available in a train and two passengers (threads) are asking for that berth in two different counters. The clerks at different counters sent a request to the server to allot that berth to their passengers. Let us see now to whom that berth is allotted.

**Program : Write a program to create multiple threads and make the threads to act on single object.**

```
//Thread unsafe –Two threads acting on same object.
class Reserve implements Runnable
{ //available berths are 1
        int available = 1;
        int wanted;
        //accept wanted berths at runtime
       Reserve (int i)
       {
              wanted = i;
        }
        public void run()
        {//display available berths
        System.out.println ("Number of berths available: " + available);
//if available berths more thanwanted betrhs
        if ( available >= wanted)
        {//get the name of the passenger
              String name = Thread.currentThread ().getName ();
```

en it encounters Thread.sleep(2000), it will suspend for specified time meanwhile t1 will

```
System.out.println (wanted + " berths alloted to: " + name);
              try
              {
                    Thread.sleep (2000); // wait for priniting the ticket
                    available = available - wanted;
```

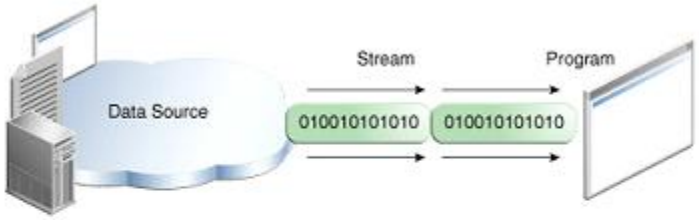
```
                    //update the no.of  available berths
                    } catch (InterruptedException ie)
                    { ie.printStackTrace ();   }
            }
            else
            {   System.out.println ("Sorry, no berths available");
            }
        }
}
class UnSafe
{
            public static void main(String args[])
            {
                    Reserve obj = new Reserve (1);
                    Thread t1 =new Thread (obj);
                    Thread t2 = new Thread (obj);
                    t1.setName ("First Person");
                    t2.setName ("Second Person");
                    t1.start ();
                    t2.start ();
            }
}
```

**I/O Streams: Stream Classes – Byte Stream Classes - Character stream classes - File Streams – Using File class – I/O Exceptions – Creation of File – Reading or writing Characters/Bytes – Random Access Files.**

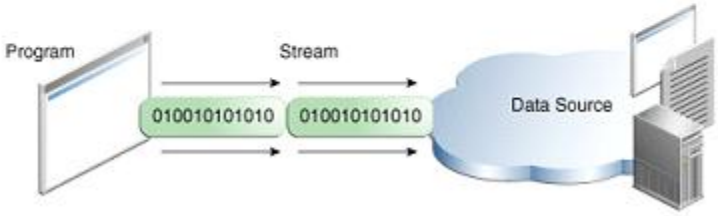
## I/O STREAMS

- So far there is no provision to store data permanently
- So go for the concept of files- data stored in files are often going to called as PERSISTENT DATA
- A file is a collection of related records placed in a particular order on the disk
- A record is composed of several fields
- A field is a group of characters
- File Processing means creation of files, updating files and manipulation of data in the file
- Reading/Writing of a file in Java can be done in the form of bytes or characters
- Reading/Writing of a file in the form of objects is called as OBJECT SERIALIZATION

A stream is a sequence of data. 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. Some streams simply pass on data; others manipulate and transform the data in useful ways. An ***I/O Stream*** represents an input source or an output destination. A program uses an *input stream* to read data from a source, one item at a time. A program uses an ***output stream*** to write data to a destination, one item at time.

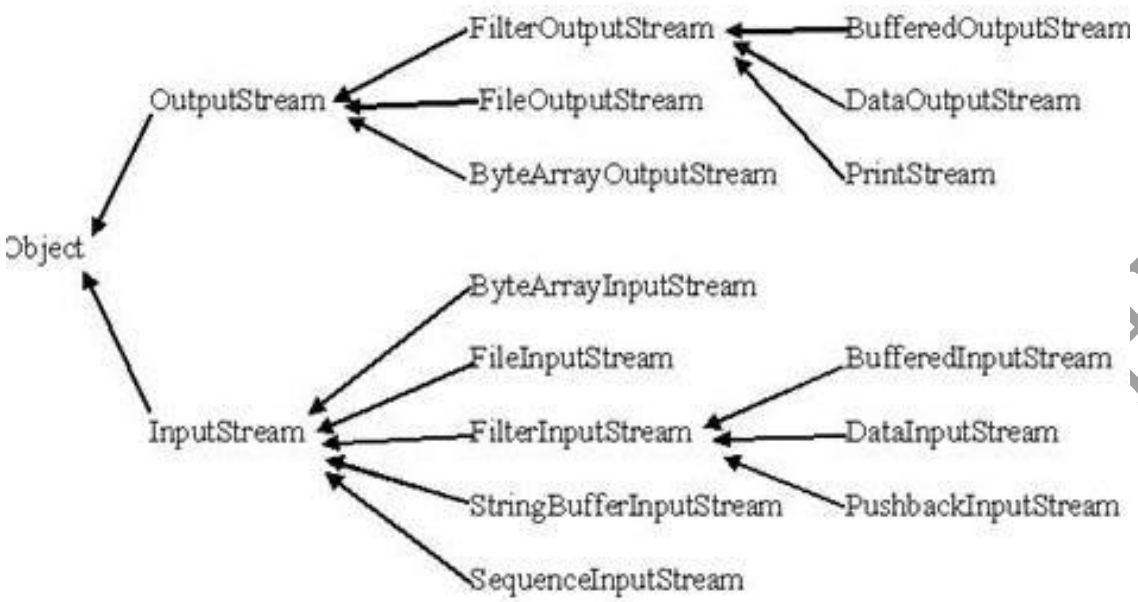



**Reading information into a program**




**Writing information from a program**

The **java.io package** contains nearly every class need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.




- Classes are categorized into two:
  Byte stream classes = Handling I/O operations on bytes
  Character stream classes = Handling I/O operations on characters
- Again there are separate input and output streams in each above said 2 classes
- Reader/writer classes are always faster than Stream classes



**LIST OF TASKS AND CLASSES**

| TASK | BYTE STREAM CLASS | CHARACTER STREAM CLASS |
|---|---|---|
| Performing Input Operations | InputStream | Reader |
| Performing output operations | OutputStream | Writer |
| Buffering Input | BufferedInputStream | BufferedReader |
| Buffering output | BufferedOutputStream | BufferedWriter |
| Keeping track of line number | LineNumberInputStream | LineNumberReader |
| Reading from an array | ByteArrayInputStream | CharArrayReader |
| Reading from files | FileInputStream | FileReader |
| Writing to a file | FileOutputStream | FileWriter |
| Reading from a pipe | PipedInputStream | PipedReader |
| Writing to a pipe | PipedOutputStream | PipedWriter |
| Reading from a string | StringBufferInputStream | StringReader |
| Writing to a string | StringBufferOutputStream | StringWriter |
| Translating byte stream into a character stream | InputStreamReader | (none) |
| Translating a character stream to a byte stream | (none) | OutputStreamWriter |

## BYTE STREAMS

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream. There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

### Using Byte Streams

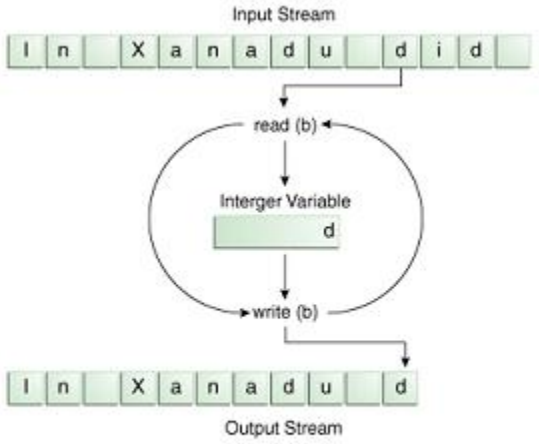
We will explore FileInputStream and FileOutputStream by examining an example program named CopyBytes, which uses byte streams to copy source.txt, one byte at a time. Byte streams should only be used for the most primitive I/O.

**Eg:**

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes
{
    public static void main(String[] args) throws IOException
{
        FileInputStream in = null;
        FileOutputStream out = null;
        try
        {
          in = new FileInputStream("source.txt");
          out = new FileOutputStream("target.txt");
          int c;
          while ((c = in.read()) != -1)
```

```
            out.write(c);
        }
      finally
      {
          in.close();
          out.close();
      }
      }
    }
```

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.




**Simple byte stream input and output.**

Notice that read() returns an int value. If the input is a stream of bytes, why doesn't read() return a byte value? Using int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.

**Reading Console Input**

- Used to read 8-bit bytes
- Class hierarchy is FileInputStream, PipeInputStream, ByteArrayInputStream, SequenceInputStream, ObjectInputStream, StringBufferInputStream, FilterInputStream, BufferedInputStream, DataInputStream, PushbackInputStream
- Some of the useful methods in InputStream are:

| | |
|---|---|
| read() | Reads a byte from the input stream |
| read(byte b[]) | Reads an array of bytes into b |
| Read(byte b[], int n, int m) | Reads m bytes into b starting from nth byte |
| available() | Gives number of bytes available in the input |
| skip(n) | Skips over n bytes from the input stream |
| reset() | Goes back to the beginning of the stream |
| close() | Closes the input stream |

Note that the class DataInputStream extends FilterInputStream and implements the interface DataInput. The methods used are: readShort(), readInt(), readLong(), readFloat(), readUTF(), readDouble(), readLine(), readChar(), readChar(),readBoolean().

OutputStream Class: OutputStream is also an abstract class and so we cannot instantiate it. It includes methods to write bytes, closing streams and flushing streams.

| Method | Description |
|---|---|
| write() | Writes a byte to the output stream |
| write(byte b[]) | Writes all bytes in the array b to the output stream |
| write(byte b[],int n,int m) | Writes m bytes from array b starting from nth byte |
| close() | Closes the output stream |
| flush() | Flushes the output stream |

Note that the class DataOutputStream extends FilterOutputStream and implements the interface DataOutput. The methods used are: writeShort(), writeInt(), writeLong(), writeFloat(), writeUTF(), writeDouble(), writeLine(), writeChar(), writeChar(),writeBoolean().

## CHARACTER STREAM CLASSES

Character stream classes were not a part of the language when it was released in 1995. Character streams can be used to read and write 16-bit Unicode characters. Here also there are two types of classes: reader and writer classes.

**Reader**: Reader stream classes are designed to read character from the files. Reader class use character as the fundamental unit of information.

**Writer**: Writer stream classes are designed to write character to the files.

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII. For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

**Using Character Streams**

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter. The CopyCharacters example illustrates these classes.

**Eg:**

```
import java.io.FileReader;
import java.io.FileWriter;
```

```
import java.io.IOException;
public class CopyCharacters
{
   public static void main(String[] args) throws IOException
 {
      FileReader inputStream = null;
      FileWriter outputStream = null;
      try
     {
         inputStream = new FileReader("f1.txt");
         outputStream = new FileWriter("f2.txt");
         int c;
         while ((c = inputStream.read()) != -1)
       outputStream.write(c);
      }
      finally
     {
         inputStream.close();
          outputStream.close();
      }
   }
}
```

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

## FILE STREAMS

**Reading and Writing Files**

As described earlier, a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination. Here is a hierarchy of classes to deal with Input and Output streams.

- Java.io package includes a class called File that provides support for file and directory manipulation
- Includes several constructors to instantiating the File objects
- Several methods for supporting operations like creating a file, opening a file, closing a file, deleting a file, getting the name of the file, getting the size of the file, checking the existence of file, renaming a file, checking whether the file is writable, checking whether the file is readable.
- First decide the following things about file

  Suitable file name = Unique string of characters with or without extension

  Data type to be stored = Data in the form of characters, bytes or primitive type

  Purpose (reading, writing or updating)

Method of creating the file

- Every file has two names as the stream object name used by Java and name used by the operating system
- A file must be opened first.
- There are 2 ways of initializing the file stream objects
  1. Provide the name of the file directly
  Eg: FileInputStream fis=new FileInputStream("input.txt")
  2. Provide the name of the file indirectly by giving the file object
  Eg:File infile=new File("input.txt")
- FileInputStream fis=new FileInputStream(infile)
- The above code includes 5 tasks:
  Select a filename
  Declare a file object
  Give the selected name to the file object declared
  Declare a file stream object
  Connect the file to the file stream object

**LIST OF EXCEPTIONS**

| I/O EXCEPTION CLASS | FUNCTIONS |
|---|---|
| EOFException | Signals that an end of the file or end of stream has been reached unexpectedly during input |
| FileNotFoundException | Informs that a file could not be found |
| InterruptedIOException | Warns that an I/O operations has been interrupted |
| IOException | Signals that an I/O exception of some sort has occured |

**COMMON STREAM CLASSES FOR I/O OPERATIONS**

| Source or Destination | Characters | | Bytes | |
|---|---|---|---|---|
| | Read | Write | Read | Write |
| Memory | CharArrayReader | CharArrayWriter | ByteArrayInputStream | ByteArrayOutputStream |
| File | FileReader | FileWriter | FileInputStream | FileOutputStream |
| Pipe | PipedReader | PipedWriter | PipedInputStream | PipedOutputStream |

**FileInputStream**

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available. Following constructor takes a file name as a string to create an input stream object to read the file.

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once we have *InputStream* object in hand then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

| No | Methods |
|---|---|
| 1 | **public void close() throws IOException{}**<br>This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}**<br>This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public int read(int r)throws IOException{}**<br>This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file. |
| 4 | **public int read(byte[] r) throws IOException{}**<br>This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned. |
| 5 | **public int available() throws IOException{}**<br>Gives the number of bytes that can be read from this file input stream. Returns an int. |

There are other important input streams available, for more detail we can refer to the following links:

- ByteArrayInputStream
- DataInputStream

**FileOutputStream**

FileOutputStream is used to create a file and write data into it.The stream would create a file, if it doesn't already exist, before opening it for output. Here are two constructors which can be used to create a FileOutputStream object. Following constructor takes a file name as a string to create an input stream object to write the file.:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once we have *OutputStream* object in hand then there is a list of helper methods which can be used to write to stream or to do other operations on the stream.

| No | Methods |
|---|---|
| 1 | **public void close() throws IOException{}**<br>This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}**<br>This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public void write(int w)throws IOException{}**<br>This methods writes the specified byte to the output stream. |
| 4 | **public void write(byte[] w)**<br>Writes w.length bytes from the mentioned byte array to the OutputStream. |

There are other important output streams available, for more detail WE can refer to the following links:

- ByteArrayOutputStream
- DataOutputStream

**Eg:** Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;
public class FileStreamTest
{  public static void main(String args[])
{    try
{
     byte bWrite [] = {11,21,3,40,5};
     OutputStream os = new FileOutputStream("C:/test.txt");
     for(int x=0; x < bWrite.length ; x++)
          os.write( bWrite[x] ); // writes the bytes
     os.close();
    InputStream is = new FileInputStream("C:/test.txt");
     int size = is.available();
    for(int i=0; i< size; i++)
         System.out.print((char)is.read() + "  ");
     is.close();
  }
catch(IOException e)
{System.out.print("Exception");}
```

```
  }
}
```

The above code would create file test.txt and would write given numbers in binary format.

## BUFFERED STREAMS

Most of the examples we've seen so far use *unbuffered* I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive. To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how WE might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

**Flushing Buffered Streams**

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer. Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format. To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

## RANDOM ACCESS FILES

- So far files are only in read only or write only mode referred as sequential files
- RandomAccessFile class files are used to create files that support both reading and writing operations
- Give the mode string as a parameter to the constructor while opening the file - “r” for reading only and “rw” for both reading and writing and also for appending

**Eg for File programs:**

```
import java.io.*;
class CopyChar
{
public static void main(String args[])
{
FileReader in=null;
FileWriter out=null;
```

```
try
{
in=new FileReader("city.txt");
out=new FileWriter("cityout.txt");
int c;
while ((c=in.read())!=-1)
  out.write((char)c);
in.close();
out.close();
}
catch(IOException e){System.out.println(e);}
}
}
```

---

```
import java.io.*;
class FileProp
{ public static void main(String args[])
{ String fname = args [0];
File f = new File (fname);
System.out.println ("File name: " + f.getName ());
System.out.println ("Path:"+ f.getPath ());
System.out.println ("Absolute Path:"+ f.getAbsolutePath ());
System.out.println ("Parent:"+ f.getParent ());
System.out.println ("Exists:"+ f.exists ());
if ( f.exists() )
{ System.out.println ("Is writable: "+ f.canWrite ());
System.out.println ("Is readable: "+ f.canRead ());
System.out.println ("Is executable: "+ f.canExecute ());
System.out.println ("Is directory: "+ f.isDirectory ());
System.out.println ("File size in bytes: "+ f.length ());
}
}
}
```

---

```
//Reading and writing bytes from a file
import java.io.*;
public class WriteRead
{public static void main(String args[])
{byte cities[]={'p','u','n','e','\n','g','o','a'};
FileOutputStream outfile=null;
FileInputStream infile=null;
int b;
try
{
outfile=new FileOutputStream("city.txt");
outfile.write(cities);
```

```
outfile.close();
infile=new FileInputStream("city.txt");
while ((b=infile.read())!=-1)
  System.out.println(b);
infile.close();
}
catch(IOException e){System.out.println(e);}
}
```

---

```
 import java.io.*;
import java.util.*;
class FileCopy
{public static void main(String a[])
{int d=0;
FileInputStream fi=null;
FileOutputStream fo=null;
Scanner s=new Scanner(System.in);
System.out.println("Enter file name:");String s1=s.nextLine();
System.out.println("Enter new file name:");String s2=s.nextLine();
try
{fi=new FileInputStream(s1);
fo=new FileOutputStream(s2);
System.out.println("Copying file");
while((d=fi.read())!=-1)
    fo.write(d);
System.out.println("File copied");
}
catch(FileNotFoundException e){System.out.println(e);}
catch(IOException e){System.out.println(e);}
}
}
```

---



## UNIT 5

**Applets – Preparing to write applets – Building Applet code – Applet life cycle – Applet tag – Passing parameters to Applets - Working with windows using AWT Classes - AWT Controls - Layout Managers and Menus.**

## JAVA APPLET

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

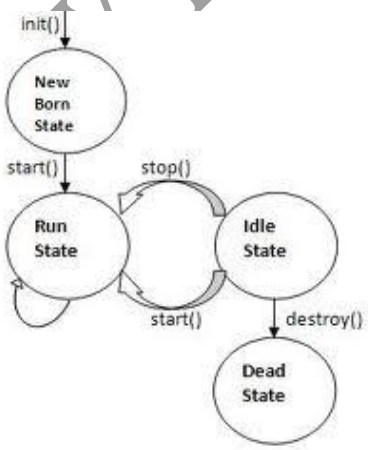
important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- 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.
- Applets have strict security rules that are enforced by the Web browser i.e., cannot read from or write to local files.

**Life Cycle of an Applet**

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

- **init (Initialization state):** 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.
  **public void init()**
  **{……**
  **……**
  **}**
- **start (Running state):** 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.
  **public void start()**
  **{……}**




- **stop (Idle state):** 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.
  **public void stop()**
  **{……**
  **……**
  **}**
- **destroy (Dead state):** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, we should not

normally leave resources behind after a user leaves the page that contains the applet.

**public void destroy()**
**{……..}**

- **paint (Display state):** 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.

**public void paint(Graphics g)**
**{…….**
**……..**
**}**

**Steps to create Applet**

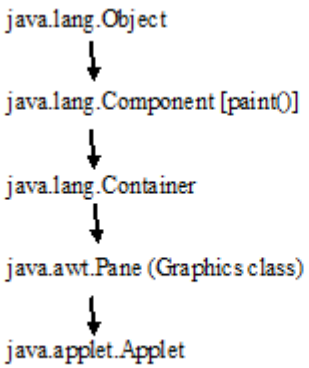
1. Building an applet code(.java file)
2. Creating an executable applet (.class file)
3. Embed the applet tag in the code
   /*<applet code=program name height=… width=….></applet>*/

**(or)**

1. Design a web page using HTML tag and write the applet tag there.
2. Save it as *.html
3. Run the program in IE or Mozilla to have results

**General format of an Applet program**

```
import java.awt.*;
import java.applet.*;
…..
…..
public class classname extends Applet
{
……
    public void paint(Graphics g)
    {……..}
}
```




**A "Hello, World" Applet**

The following is a simple applet named HelloWorldApplet.java:

```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet
{
  public void paint (Graphics g)
  {
    g.drawString ("Hello World", 25, 50);
    g.showStatus("This will be shown in status bar");
  }
}
```

These import statements bring the classes into the scope of our applet class:

- java.applet.Applet.
- java.awt.Graphics.

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to. Save the file just like with Java applications, give our file a name that has the same name as the class. In this case, the filename would be "HelloWorldApplet.java". The paint() method displays the content of the applet onto the screen. Here, the string Hello World gets drawn. Applets use several standard methods to take the place of main(), which include init() to initialize the applet, start(0 to start it running , and paint() to display it to the screen. Now, compile the applet just as we did the application, using javac, the Java compiler.

>javac HelloWorldApplet.java

Again, just as for application, we should now have a file called HelloWorldApplet.class in our directory.

To include an applet in a Web page, we refer to that applet in the HTML code for that Web page. Here, we create a very simple HTML
file in the directory.

```
<HTML>
<HEAD>
<TITLE>
Hello to Everyone!
</TITLE>
</HEAD>
<BODY>
<P> My Java applet says:
<APPLET CODE=.HelloWorldApplet.class. WIDTH = 150 HEIGHT = 25>
</APPLET>
</BODY>
</HTML>
```

**Sample Applet Programs**

```
import java.awt.*;
import java.applet.*;
public class Appletexec extends Applet
```

```
{
public void init()
{System.out.println("I am inside init");}
public void start()
{System.out.println("I am inside start");}
public void stop()
{System.out.println("I am inside stop");}
public void destroy()
{System.out.println("I am inside destroy");}
public void paint(Graphics g)
{System.out.println("I am inside paint");}
}
/*<applet code=Appletexec width=500 height=500></applet>*/
```

---

```
/* <applet code="MyApplet.class" width = 600 height= 450></applet> */
import java.applet.Applet;
import java.awt.*;
public class MyApplet extends Applet
{ String msg="";
public void init()
{msg += "init";}
public void start()
{msg +=" start";}
public void paint(Graphics g)
{g.drawString(msg,10,100);}
public void stop()
{msg += " stop";}
public void destroy()
{msg+= " destroy";}
}
```

---

```
/* <applet code="MyApplet2.class" width = 600 height= 450>
<param name = "t1" value="102">
<param name = "t2" value ="101">
</applet> */
import java.applet.*;
import java.awt.*;
public class MyApplet2 extends Applet
{ int n1,n2,n;
public void init()
{ n1 = Integer.parseInt(getParameter("t1"));n2 = Integer.parseInt(getParameter("t2"));n=n1+n2;}
public void paint(Graphics g)
{ g.drawString(String.valueOf(n),100,150);}
}
```

---

```
import java.awt.*;
```

```
import java.applet.*;
public class Param extends Applet
{
String str;
public void init()
{str=getParameter("string");}
public void paint(Graphics g)
{g.drawString(str,10,10);}
}
/* <applet code=Param.class width=500 height=5500>
<param name="string" value="welcome">
</applet>*/
```

```
/* <applet code="MyApplet2.class" width = 600 height= 450>
<param name = "t1" value="Hari Prasad">
<param name = "t2" value ="101">
</applet> */
import java.applet.*;
import java.awt.*;
public class MyApplet2 extends Applet
{ String n;
String id;
public void init()
{ n = getParameter("t1");id = getParameter("t2");}
public void paint(Graphics g)
{ g.drawString("Name is : " + n, 100,100);g.drawString("Id is : "+ id, 100,150);}
```



## DRAWING SHAPES

Most of the drawing methods require us to specify a bounding rectangle for the object we want to draw: the location of the object's upper left corner, plus its width and height. The two exceptions are lines and polygons. For lines, we need to supply two endpoints; for polygons, we need to provide a set of points.

1. **public abstract void drawLine (int x1, int y1, int x2, int y2)**

The drawLine() method draws a line on the graphics context in the current color from (x1, y1) to (x2, y2). If (x1, y1) and (x2, y2) are the same point, it will draw a point.

```
g.drawLine (5, 5, 50, 75); // line
g.drawLine (5, 75, 5, 75); // point
g.drawLine (50, 5, 50, 5); // point
```

2. **public void drawRect (int x, int y, int width, int height)**

The drawRect() method draws a rectangle on the drawing area in the current color from (x, y) to (x+width, y+height). If width or height is negative, nothing is drawn.

3. **public abstract void fillRect (int x, int y, int width, int height)**

The fillRect() method draws a filled rectangle on the drawing area in the current color from (x, y) to (x+width-1, y+height-1). Notice that the filled rectangle is one pixel

smaller to the right and bottom than requested. If width or height is negative, nothing is drawn.

4. **public abstract void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)**

The drawRoundRect() method draws a rectangle on the drawing area in the current color from (x, y) to (x+width, y+height). However, instead of perpendicular corners, the corners are rounded with a horizontal diameter of arcWidth and a vertical diameter of arcHeight. If width or height is a negative number, nothing is drawn. If width, height, arcWidth, and arcHeight are all equal, we get a circle.

5. **public abstract void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)**

The fillRoundRect() method draws a filled rectangle on the drawing area in the current color from (x, y) to (x+width-1, y+height-1). However, instead of having perpendicular corners, the corners are rounded with a horizontal diameter of arcWidth and a vertical diameter of arcHeight for the four corners. Notice that the filled rectangle is one pixel smaller to the right and bottom than requested. If width or height is a negative number, nothing is filled. If width, height, arcWidth, and arcHeight are all equal, we get a filled circle.

```
g.drawRect (25, 10, 50, 75);
g.fillRect (25, 110, 50, 75);
g.drawRoundRect (100, 10, 50, 75, 60, 50);
g.fillRoundRect (100, 110, 50, 75, 60, 50);
```

6. **public abstract void drawOval (int x, int y, int width, int height)**

The drawOval() method draws an oval in the current color within an invisible bounding rectangle from (x, y) to (x+width, y+height). We cannot specify the oval's center point and radii. If width and height are equal, we get a circle. If width or height is negative, nothing is drawn.

7. **public abstract void fillOval (int x, int y, int width, int height)**

The fillOval() method draws a filled oval in the current color within an invisible bounding rectangle from (x, y) to (x+width-1, y+height-1). We cannot specify the oval's center point and radii. Notice that the filled oval is one pixel smaller to the right and bottom than requested. If width or height is negative, nothing is drawn.

```
g.drawOval (25, 10, 50, 75);
g.fillOval (25, 110, 50, 75);
g.drawOval (100, 10, 50, 50);
g.fillOval (100, 110, 50, 50);
```

8. **public abstract void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)**

The drawArc() method draws an arc in the current color within an invisible bounding rectangle from (x, y) to (x+width, y+height). The arc starts at startAngle degrees and goes to startAngle + arcAngle degrees. An angle of 0 degrees is at the 3 o'clock position; angles increase counter-clockwise. If arcAngle is negative, drawing is in a clockwise direction. If width and height are equal and arcAngle is 360 degrees, drawArc() draws a circle. If width
or height is negative, nothing is drawn.

**9. public abstract void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)**

The fillArc() method draws a filled arc in the current color within an invisible bounding rectangle from (x, y) to (x+width-1, y+height-1). The arc starts at startAngle degrees and goes to startAngle + arcAngle degrees. An angle of 0 degrees is at the 3 o'clock position; angles increase counter-clockwise. If arcAngle is negative, drawing is in a clockwise direction. The arc fills like a pie (to the origin), not from arc endpoint to arc endpoint. This makes creating pie charts easier. If width and height are equal and arcAngle is 360 degrees, fillArc() draws a filled circle. If width or height is negative, nothing is drawn.

```
g.drawArc (25, 10, 50, 75, 0, 360);
g.fillArc (25, 110, 50, 75, 0, 360);
g.drawArc (100, 10, 50, 75, 45, 215);
g.fillArc (100, 110, 50, 75, 45, 215);
```

**10. public void drawPolygon (Polygon p)**

The drawPolygon() method draws a path for the points in polygon p in the current color.

**11. public abstract void drawPolygon (int xPoints[], int yPoints[], int numPoints)**

The drawPolygon() method draws a path of numPoints nodes by plucking one element at a time out of xPoints and yPoints to make each point.

```
int[] xPoints[] = {{50, 25, 25, 75, 75}, {50, 25, 25, 75, 75}, {100, 100, 150, 100, 150, 150, 125, 100, 150},
{100, 100, 150, 100, 150, 150, 125, 100, 150}};
int[] yPoints[] = {{10, 35, 85, 85, 35, 10},
{110, 135, 185, 185, 135},
{85, 35, 35, 85, 85, 35, 10, 35, 85},
{185, 135, 135, 185, 185, 135, 110, 135, 185}};
int nPoints[] = {5, 5, 9, 9};
g.drawPolygon (xPoints[0], yPoints[0], nPoints[0]);
g.fillPolygon (xPoints[1], yPoints[1], nPoints[1]);
g.drawPolygon (new Polygon(xPoints[2], yPoints[2], nPoints[2]));
g.fillPolygon (new Polygon(xPoints[3], yPoints[3], nPoints[3]));
```

**Eg:**

```
import java.applet.*;
import java.awt.*;
public class CircleLine extends Applet{
int x=300,y=100,r=50;
public void paint(Graphics g){
g.drawLine(3,300,200,10);
g.drawString("Line",100,100);
g.drawOval(x-r,y-r,100,100);
g.drawString("Circle",275,100);
g.drawRect(400,50,200,100);
g.setColor(Color.red);
g.drawString("Rectangle",450,100);
}
```

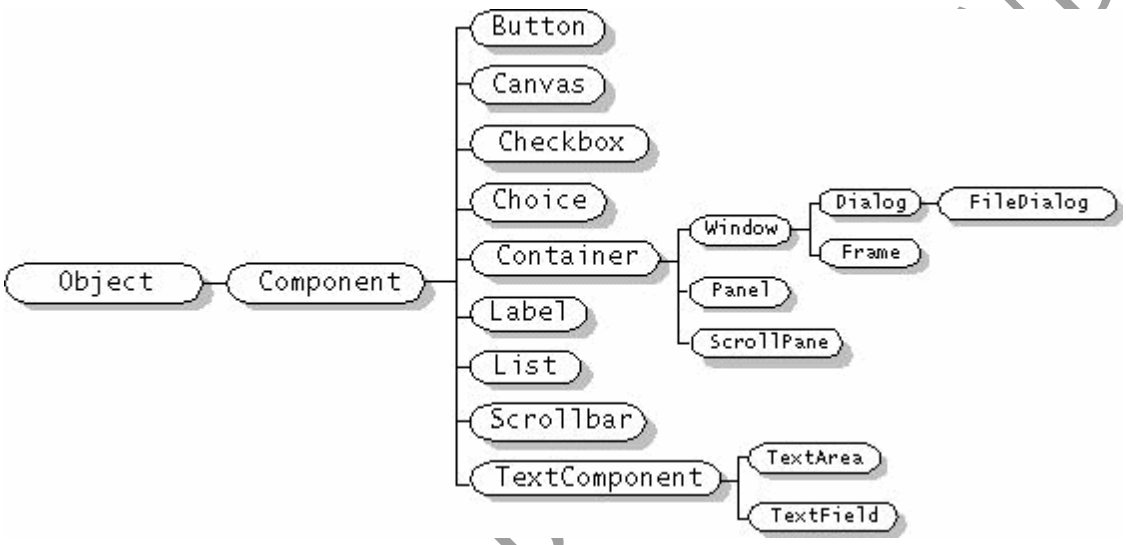
}

## AWT CONTROLS

The AWT was designed from the start to have a platform-independent API and yet to preserve each platform's look and feel. For example, the AWT has just one API for buttons (provided by the Button class), but a button looks different on a Macintosh than on a PC running Windows 95. The AWT achieves its seemingly contradictory goals by providing classes (*components*) that provide a platform-independent API but that make use of platform specific implementations (*peers*).

### Hierarchy of AWT Components

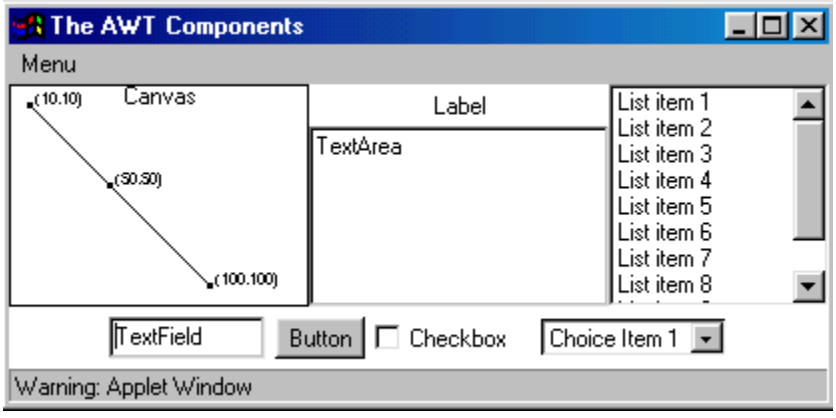
The following diagram shows the hierarchy of awt components




### AWT Components

AWT Components are Java's building blocks for creating graphical user interface. Some component types, such as buttons and scroll bars, are used directly for GUI control. Other components provide spatial organization. GUI is an important part of any program. Java's **A**bstract **W**indowing **T**oolkit provides extensive functionality. Some of the important components are

1. Buttons.
2. Checkboxes
3. Choices
4. Labels
5. Panels
6. Scrollbars
7. Textarea and Textfield



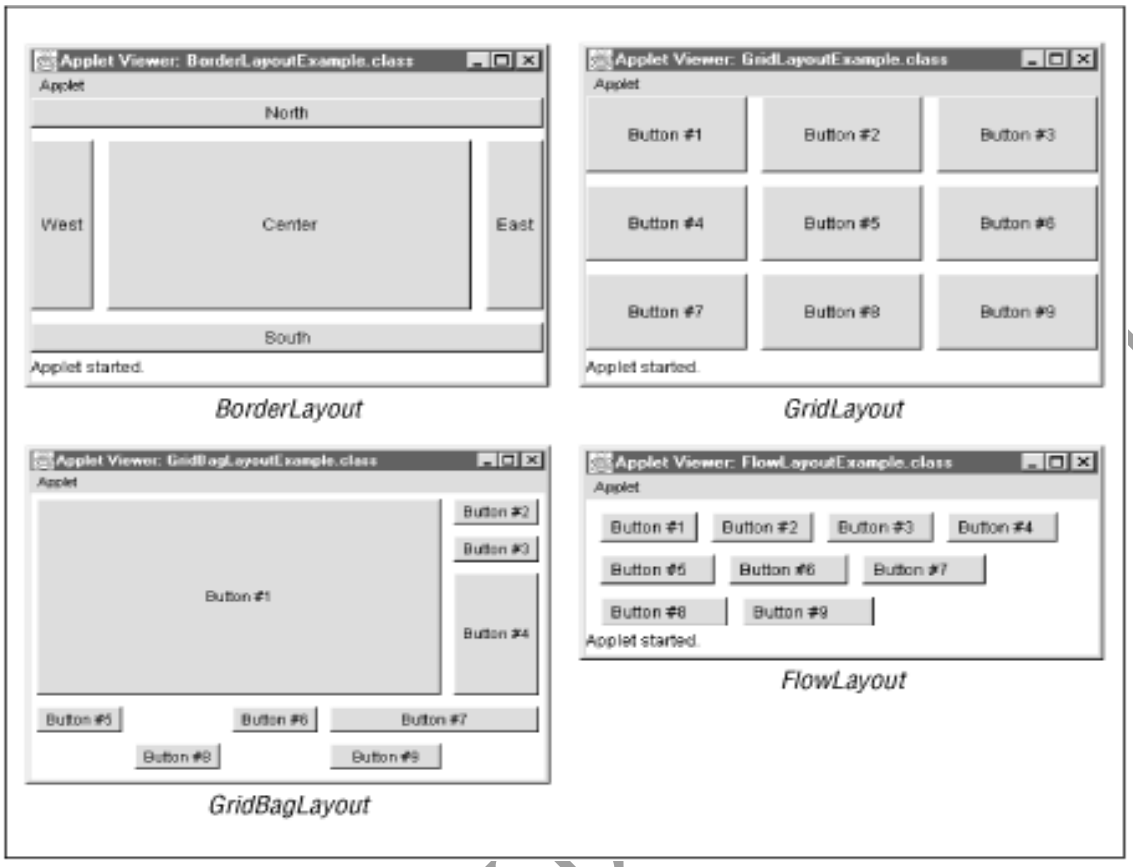
Java's components are implemented by many subclasses of the java.awt.Component class and java.awt.MenuComponent class. There are 19 non-super class components in all. Let us look at the basic if these components. Components can be broadly classified as:

1. Visual Component (Button, CheckBox etc)
2. Container Components (Frame, Panel, Applet etc)
3. Menu Component (Menu, MenuItem, MenuBar etc)

**Layout Managers**

A layout manager is an object that controls the size and position (layout) of components inside a Container object. For example, a window is a container that contains components such as buttons and labels. The layout manager in effect for the window determines how the components are sized and positioned inside the window. A container can contain another container. For example, a window can contain a panel, which is itself a container. As we can see in the figure, the layout manager in effect for the window determines how the two panels are sized and positioned inside the window, and the layout manager in effect for each panel determines how components are sized and positioned inside the panels.

| Layout Manager | Description |
|---|---|
| BorderLayout | Layout a maximum of five components: one along each of the four borders of the container and one in the center. When using this layout manager, we must add components to the container using a two-argument version of add() method. The constraint argument should be one of the strings "North", "East", "South", "West", or "Center". Despite the simplicity of this layout system, this layout manager is used quite often. |
| CardLayout | Makes each component as large as the container and displays only one at a time. Various methods change the currently displayed component. |
| FlowLayout | Arranges components like words on a page: from left to right in rows and then top to bottom as each row fills up. Rows may be left, center, or right justified. |
| GridBagLayout | A flexible layout manager that arranges components in a grid with variable-sized cells. Allows explicit control over the way each component is resized when the container changes size. Requires a complex constraints set specified with the GridBagConstraints object. |
| GridLayout | Makes all components the same size and arranges them in a grid of specified dimensions. |




**Flow Layout**

```
import java.applet.Applet;
import java.awt.*;
/* Class illustrates simple Flow Layout of three components.
Since Flow Layout is the default one for Applets (and Panels),
it does not have to be set explicitly here.  */
public class Flow1 extends Applet
{
 public void init()
 {
// if we want to explicitly set the Flow Layout Manager we would type
//  setLayout ( new FlowLayout() );  or...
//  this.setLayout ( new FlowLayout() );
   Label label = new Label("Name:");      // create a Label
   add(label);                            // add it
   TextField tf = new TextField("Java");   // create a TextField
   add(tf);                               // add it
   Button b = new Button("OK");           // create a Button
   add(b);                                // and add it
 }
}
```

**Grid Layout**

```
import java.applet.Applet;
import java.awt.*;
public class Grid extends Applet
{
 public void init()
 {
  // we must explicitly set GridLayout as the manager
  setLayout (new GridLayout(5, 3));    // 5 rows, 3 columns, no gaps
  for (int row = 0; row < 5; row ++)
  {
   add (new Label("Label " + row));
   add (new Button("Button " + row));
   add (new TextField("TextField " + row));
  }
 }
}
```

**Border Layout**

First example puts components at North, West, and East.

```
import java.applet.Applet;
import java.awt.*;
public class Border1 extends Applet
{
 public void init()
 {
  setLayout (new BorderLayout());
  Label l = new Label("This is NORTH", Label.CENTER);
  l.setBackground(Color.yellow);
  Scrollbar sb1 = new Scrollbar();
  Scrollbar sb2 = new Scrollbar();
  add(l, BorderLayout.NORTH);
  add(sb1, BorderLayout.WEST);
  add(sb2, BorderLayout.EAST);
 }
}
```

**Card Layout**

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
/* This applet demonstrates using the CardLayout manager.
Pressing one of three buttons will cause a different "card" to be
displayed.
*/
public class Card1 extends Applet implements ActionListener
{
 Panel cardPanel;       // the container that will hold the various "cards"
 Panel firstP, secondP, thirdP;  // each of these panels will constitute the "cards"
```

```
 Panel buttonP;              // panel to hold three buttons
 Button first, second, third;   // the three buttons
 CardLayout ourLayout;         // the card layout object
  public void init()
 {
  //create cardPanel which is the panel that will contain the three "cards"
  cardPanel = new Panel();
  //create the CardLayout object
  ourLayout = new CardLayout();
  //set card Panel's layout to be our Card Layout
  cardPanel.setLayout (ourLayout);

//create three dummy panels (the "cards") to show
  firstP = new Panel();
  firstP.setBackground(Color.blue);

  secondP = new Panel();
  secondP.setBackground(Color.yellow);

  thirdP = new Panel();
  thirdP.setBackground(Color.green);

  //create three buttons and add ActionListener
  first = new Button("First");
  first.addActionListener(this);

  second = new Button("Second");
  second.addActionListener(this);

  third = new Button("Third");
  third.addActionListener(this);

  //create Panel for the buttons and add the buttons to it
  buttonP = new Panel();   // Panel's default Layout manager is FlowLayout
  buttonP.add(first);
  buttonP.add(second);
  buttonP.add(third);

  //setLayout for applet to be BorderLayout
  this.setLayout(new BorderLayout());
  //button Panel goes South, card panels go Center
  this.add(buttonP, BorderLayout.SOUTH);
  this.add(cardPanel, BorderLayout.CENTER);

  // add the three card panels to the card panel container
  // method takes 1.) an Object (the card) 2.) an identifying String
```

```
  // first one added is the visible one when applet appears
  cardPanel.add(firstP, "First");     //blue
  cardPanel.add(secondP, "Second");   //yellow
  cardPanel.add(thirdP, "Third");     //green
   }

 // respond to Button clicks by showing the so named Panel
 // note use of the CardLayout method show(Container, "identifying string")

 public void actionPerformed(ActionEvent e)
 {
  if (e.getSource() == first)
   ourLayout.show(cardPanel, "First");

  if (e.getSource() == second)
   ourLayout.show(cardPanel, "Second");

  if (e.getSource() == third)
   ourLayout.show(cardPanel, "Third");
 }
} // end class
```

**Frames**

We can create a Frame by creating Frame class object.

Frame f = new Frame ();

(or)

Create a class that extends Frame class then create an object to that class as

```
class MyClass extends Frame
MyClass f= new MyClass ();
```

- After creating the Frame we need to set Frame width and height using setSize () method as:

  f.setSize (400, 350);

- We can display the frame using setVisible () method as:

  f.setVisible (true);

- We can have the title for the frame as:

  f.setTitle("Title");

**Eg:**

```
//Creating a Frame
import java.awt.*;
class MyFrame
{
public static void main(String args[])
{ Frame f1 = new Frame ();
f1.setSize (500,150);
f1.setTitle ("GUI World");
f1.setVisible (true);
```

```
}
}
```

**(or)**

```
//creating a Frame
import java.awt.*;
class MyFrame extends Frame
{ public static void main(String args[])
{ MyFrame f1 = new MyFrame ();
f1.setSize (500,200);
f1.setTitle ("GUI World");
f1.setVisible (true);
}}
```

The frame can be minimized, maximized and resized but cannot be closed. Even if we click on close button of the frame, it will not perform any closing action. Closing a frame means attaching action to the component. To attach actions to the components, we need 'Event Delegation Model'. The listener interfaces in Java are:

- ✖ Window Listener
- ✖ Action Listener
- ✖ Focus Listener
- ✖ Adjustment Listener

The methods available in WindowListener are:

```
public void windowActivated (WindowEvent e)
public void windowDeactivated (WindowEvent e)
public void windowOpened (WindowEvent e)
public void windowClosed (WindowEvent e)
public void windowClosing (WindowEvent e)
public void windowIconified (WindowEvent e)
public void windowDeiconified (WindowEvent e)
```

We should attach a listener to the frame component. Remember, all listeners are available in java.awt.event package. The most suitable listener to the frame is 'WindowListener'. It can be attached using addWindowListener () method as: f.addWindowListener (WindowListener obj); and implement all the methods of the WindowListener interface.

**Eg:**

```
//creating a Frame and closing the same
import java.awt.*;
import java.awt.event.*;
public class MyFrame extends Frame implements WindowListener
{ public static void main(String args[])
{ MyFrame f1 = new MyFrame ();
f1.setSize (500,200);
f1.setTitle ("GUI World");
f1.setVisible (true);
f1.addWindowListener (new MyFrame());
```

```
}
public void windowClosing (WindowEvent e)
{
System.exit (0);
}
public void windowActivated (WindowEvent e){}
public void windowClosed (WindowEvent e){}
public void windowDeactivated (WindowEvent e){}
public void windowDeiconified (WindowEvent e){}
public void windowIconified (WindowEvent e){}
public void windowOpened (WindowEvent e){}
}
```

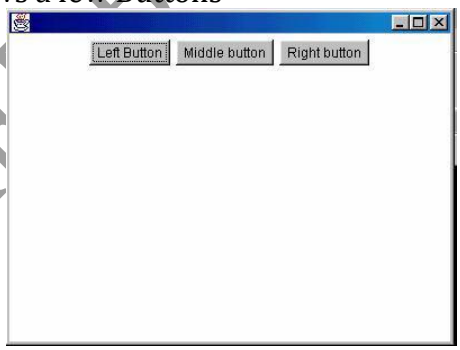
**Buttons**

A Button is a simple control that generates an action event when the user clicks it. The onscreen appearance of Buttons depends on the platform they're running on and on whether the Button is enabled. The only facets of a Button's appearance that we can change without creating our own class are the font and text it displays, its foreground and background colors, and (by enabling or disabling the button) whether the Button looks enabled or disabled.

- To create push button: Button b=new Button("label");
- To get the label of the button: String l = b1.getLabel();
- To set the label of the button: b.setLabel("label");
- To get the label of the button clicked: String str = ae.getActionCommand();

where ae is object of ActionEvent.

**Eg:** The following figure shows a few Buttons




```
import java.awt.*;
//The following class demonstrate use of Button
public class ButtonDemo extends Frame
{
Button b1 = new Button("Left Button");
Button b2 = new Button("Middle button");
Button b3 = new Button("Right button");
public ButtonDemo()
{
setLayout(new FlowLayout());
```

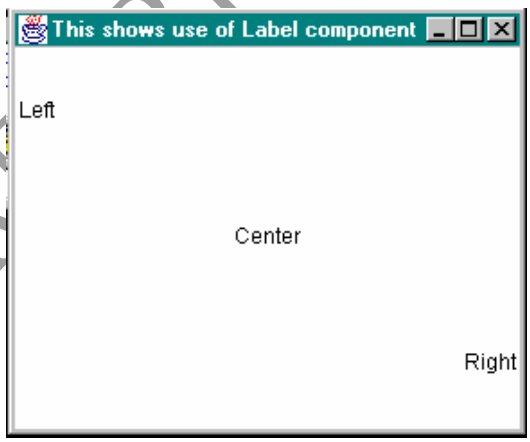
```
add(b1);
add(b2);
add(b3);
}
public static void main(String[] args)
{
ButtonDemo bdemo = new ButtonDemo();
bdemo.setSize(400,300);
bdemo.setVisible(true);
}
}
```

**Labels**

The Label class provides an easy way of putting unselectable text in our program's GUI. Labels are aligned to the left of their drawing area, by default. We can specify that they be centered or right-aligned by specifying Label.CENTER or Label.RIGHT either to the Label constructor or to the setAlignment() method. As with every Component, we can also specify the font and color of a Label. Label Example There are three labels, each one with a different alignment. If each label's display area were equal to the width of the text the label displayed, we wouldn't see any difference in the alignment of the labels. Each label's text would simply be displayed using all the available space.

- To create a label: Label l = new Label("text",alignmentConstant);

Note: - alignmentconstant: Label.RIGHT, Label.LEFT and Label.CENTER




**Eg:**

```
import java.awt.*;
import java.awt.event.*;
public class LabelDemo extends Frame
{
Label l1 = new Label("Left");
Label l2 = new Label("Center");
Label l3 = new Label("Right");
public LabelDemo()
```

```
{
setTitle("This shows use of Label component");
setSize(200,100);
setLayout(new GridLayout(3,1));
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we)
{
dispose();
System.exit(0);
}
});
l1.setAlignment(l1.LEFT);
l2.setAlignment(l2.CENTER);
l3.setAlignment(l3.RIGHT);
add(l1);
add(l2);
add(l3);
}
public static void main(String s[])
{
LabelDemo ld = new LabelDemo();
ld.setSize(200,200);
ld.show();
}
}
```

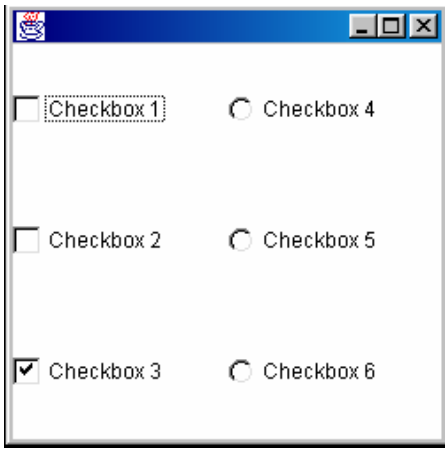
**Checkboxes**

- When the user clicks a checkbox, the Checkbox State changes and it generates an action event. Other ways of providing groups of items the user can select are choices, lists, and menus. If we want a group of checkboxes in which only one checkbox at a time can be "on", we can add a CheckboxGroup object to oversee the state of the checkboxes. (We call this element as a *radio button*.) Below is an example that has two columns of checkboxes. On the left are three independent checkboxes. We can select all three of the checkboxes, if we like. On the right are three checkboxes that are coordinated by a CheckboxGroup object. The CheckboxGroup ensures that no more than one of its checkboxes is selected at a time. To be specific, a checkbox group can come up with no checkboxes selected, but once the user selects a checkbox, exactly one of the checkboxes will be selected forever after. Following is the code that creates both groups of checkboxes. Note that only the second, mutually exclusive group of checkboxes is controlled by a CheckboxGroup.
- To create a Checkbox: Checkbox cb = new Checkbox ("label");
- To create a checked Checkbox: Checkbox cb = new Checkbox ("label", null, true);
- To get the state of a Checkbox: boolean b = cb.getState ();
- To set the state of a Checkbox: cb.setState (true);
- To get the label of a Checkbox: String s = cb.getLabel ();

**Eg:**

```
import java.awt.*;
public class CheckBoxDemo extends Frame
{
Panel p1, p2;
Checkbox cb1, cb2, cb3; //These are independent checkboxes.
Checkbox cb4, cb5, cb6; //These checkboxes are part of a group.
CheckboxGroup cbg;
public CheckBoxDemo()
{
cb1 = new Checkbox(); //Default state is "off"
(false).
cb1.setLabel("Checkbox 1");
cb2 = new Checkbox("Checkbox 2");
cb3 = new Checkbox("Checkbox 3");
cb3.setState(true); //Set state to "on" (true).
cbg = new CheckboxGroup();
cb4 = new Checkbox("Checkbox 4", cbg, false); //initial
state: off
cb5 = new Checkbox("Checkbox 5", cbg, false); //initial
state: off
cb6 = new Checkbox("Checkbox 6", cbg, false); //initial
state: off
setLayout(new GridLayout(3,2));
add(cb1);add(cb4);
add(cb2);add(cb5);
add(cb3);add(cb6);
}
public static void main(String[] args)
{
CheckBoxDemo cb = new CheckBoxDemo();
cb.setSize(250,250);
cb.setVisible(true);
}
}
```

Besides the Checkbox methods shown above, Checkbox has two additional methods we might want to use: getCheckboxGroup() and setCheckboxGroup(). Besides the single CheckboxGroup constructor used in the code example, CheckboxGroup also defines the following methods: getCurrent() and setCurrent(). These methods get and set (respectively) the current selected Checkbox. Note: CheckboxGroup is only a helper class, it is not a component by itself. Its basic use is to generate RadioButtons. The program in a gridlayout would look similar to this:

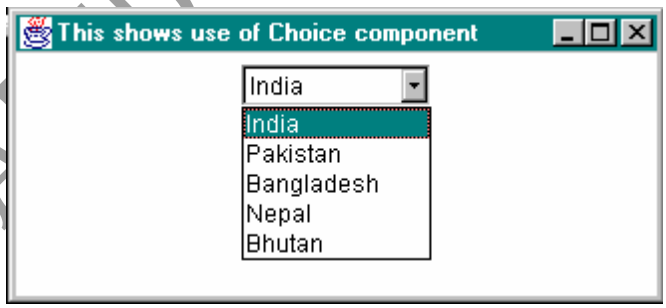



**Choices**

The Choice class provides a menu-like list of choices, accessed by a distinctive button. The user presses the button to bring up the "menu", and then chooses one of the items. When the user chooses an item, the Choice generates an action event. Choices are useful when we need to display a number of alternatives in a limited amount of space, and the user doesn't need to see all the alternatives all the time. Another name we might know for this UI element is *pop-up list*. Other ways of providing multiple alternatives are checkboxes, lists, and menus. Following is an example of a Choice and a Label.

- To create a choice menu: Choice ch = new Choice();
- To add items to the choice menu: ch.add ("text");
- To know the name of the item selected from the choice menu: String s = ch.getSelectedItem ();
- To know the index of the currently selected item: int i = ch.getSelectedIndex();

This method returns -1, if nothing is selected.




```
import java.awt.*;
public class ChoiceDemo extends Frame
{
Choice ch = new Choice();
public ChoiceDemo()
{
setTitle("This shows use of Choice component");
setSize(200,100);
setLayout(new FlowLayout());
ch.addItem("India");
```

```
ch.addItem("Pakistan");
ch.addItem("Bangladesh");
ch.addItem("Nepal");
ch.addItem("Bhutan");
add(ch);
}
public static void main(String[] args)
{
ChoiceDemo cd = new ChoiceDemo();
cd.show();
}
}
```

Besides the methods used above, the Choice class defines these other useful methods:

1. **int countItems()=**Returns the number of items in the choice.
2. **String getItem(int)=**Returns the String displayed by the item at the specified index.
3. **void select(int)=**Selects the item at the specified index.
4. **void select(String)=**Selects the item that's displaying the specified String.

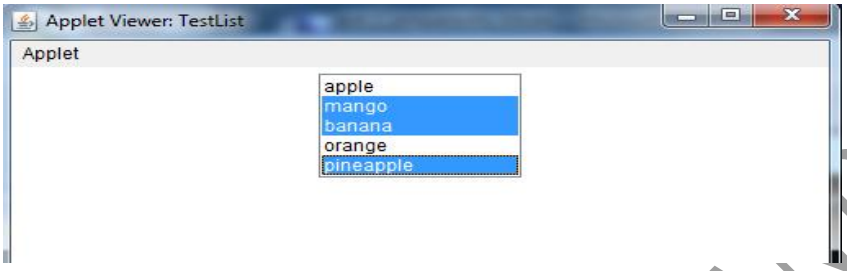
**Listbox**

- A List box is similar to a choice box, it allows the user to select multiple items.
- To create a list box: List lst = new List();
  (or)
  List lst = new List (3, true);
- This list box initially displays 3 items. The next parameter true represents that the user can select more than one item from the available items. If it is false, then the user can select only one item.
- To add items to the list box: lst.add("text");
- To get the selected items: String x[] = lst.getSelectedItems();
- To get the selected indexes: int x[] = lst.getSelectedIndexes ();
- **list.replaceItem("Linux",0):** Replaces the item at the first position with Linux
- **list.deleteItem(2):** Deletes the element residing at index 2
- **list.deleteItems(5,10):** Delete all the items from index 5 to 10
- **list.clear():** Deletes all the items from the list

**Eg:**

```
import java.awt.*;
import java.applet.*;
public class TestList extends Applet
{
  public void init()
{
        List l = new List(5, true); //prefer 5 items visible
        l.add("apple");
        l.add("mango");
        l.add("banana");
        l.add("orange");
```

```
        l.add("pineapple");
        add(l);
    }
}
//<applet code=TestList width=500 height=600></applet>
```




**Textarea and Textfield**

The TextArea and TextField classes display selectable text and, optionally, allow the user to edit the text. We can subclass TextArea and TextField to perform such tasks as checking for errors in the input. As with any Component, we can specify the background and foreground colors and font used by TextAreas and TextFields. we can't, however, change their basic appearance. Both TextArea and TextField are subclasses of TextComponent. From TextComponent they inherit methods that allow them to set and get the current selection, enable and disable editing, get the currently selected text (or all the text), and set the text.

- To create a TextField: TextField tf = new TextField(25);

(or)

TextField tf = new TextField ("defaulttext", 25);

- To get the text from a TextField: String s = tf.getText();
- To set the text into a TextField: tf.setText("text");
- To hide the text being typed into the TextField by a character: tf.setEchoChar('char');
- To edit the text entered in the textboxes: tf.setEditable(true);
- To create a TextArea: TextArea ta = new TextArea();

(or)

TextArea ta = new TextArea (rows, cols);

Note: TextArea supports getText () and setText ()


//Where instance variables are defined:

```
TextField textField;
TextArea textArea;
public void init() {
textField = new TextField(20);
textArea = new TextArea(5, 20);
...//Add the two components to the panel.
}
```

The TextComponent superclass of TextArea and TextField supplies the getText(), setText(), setEditable(), and selectAll() methods used in the above code example. It also supplies the following useful methods: getSelectedText(), isEditable(), getSelectionStart(), and getSelectionEnd(). It also provides a select() method that lets us select text between beginning and end positions that we specify.

1. **int getColumns()=**Returns the number the columns in the text field.
2. **setEchoChar()=**Sets the echo character, which is useful for password fields.
3. **char getEchoChar() boolean echoCharIsSet()=**These methods let us ask about the echo character. Like the TextField class, the TextArea class also has four constructors: TextArea(), TextArea(int, int), TextArea(String), and TextArea(String, int, int). The integer arguments specify the number of rows and columns (respectively) in the text area. The String argument specifies the text initially displayed in the text area. The TextArea class supplies the appendText() method used in the code example above. It also supplies these methods:
4. **int getRows(), int getColumns()=**Return the number of rows or columns in the text area.
5. **void insertText(String, int)=**Inserts the specified text at the specified position.
6. **void replaceText(String, int, int)=**Replaces text from the indicated start position (the first integer) to the indicated end position.

**Scroll Bar**

- Scrollbar class is useful to create scrollbars that can be attached to a frame or text area
- Scrollbars can be arranged vertically or horizontally.
- To create a scrollbar : Scrollbar sb = new Scrollbar (alignment, start, step, min, max);
  where alignment: Scrollbar.VERTICAL, Scrollbar.HORIZONTAL
  start: starting value (e.g. 0)
  step: step value (e.g. 30)
  min: minimum value (e.g. 0)
  max: maximum value (e.g. 255)
- To know the location of a scrollbar: int n = sb.getValue ();
- To update scrollbar position to a new position: sb.setValue (int position);
- To get the maximum value of the scrollbar: int x = sb.getMaximum ();
- To get the minimum value of the scrollbar: int x = sb.getMinimum ();
- To get the alignment of the scrollbar: int x = getOrientation ();

This method return 0 if the scrollbar is aligned HORIZONTAL, 1 if aligned VERTICAL.

**Eg:**

```
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
public class TestScroll extends Applet implements AdjustmentListener
{
Scrollbar s;
int red=255;
Label l1;
public TestScroll()
{
s = new Scrollbar (Scrollbar.VERTICAL, 0, 50, 10,255 );
l1=new Label(String.valueOf(red));
add (s);add(l1);
s.addAdjustmentListener (this);
}
public void adjustmentValueChanged(AdjustmentEvent e)
{
int v=e.getValue();
l1.setText(String.valueOf(v));
repaint();
}
public void paint(Graphics g)
{
g.setColor(new Color(red,0,180));
g.fillOval(40,50,60,60);
}
}
//<applet code=TestScroll width=400 height=400></applet>
```




**Menus**

As a name indicates a Menu consists of Menu objects. These Menu objects comprise of MenuItem objects which can be selected by the user with a click of a mouse. A MenuItem may be a String, checkbox, separator, menu etc. Following are the steps to add menus to any Frame:

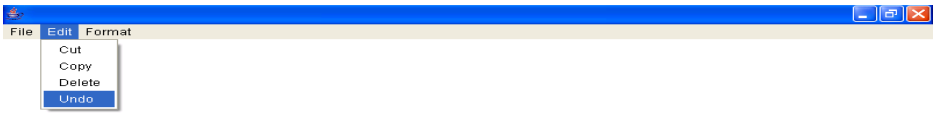
1. Create a MenuBar first with the help of the following method.
   MenuBar mb = new MenuBar();
2. Then create a Menu using Menu m = new Menu("File");.
3. Now the MenuItem options can be added to the Menu from top to bottom, using the following methods.

```
        mi.add(new MenuItem("Open"));
        mi.add(new CheckboxMenuItem("Type here"));
```

4. Add the Menu to the MenuBar from left to right using mi.add(m);.
5. Finally, add the MenuBar to the Frame by calling the setMenuBar() method.

The program code given below creates an application window with a menu bar.

```
public class TestMenu extends Frame
{
public TestMenu()
 {
  MenuBar m=new MenuBar();
 Menu  m1=new Menu("File");   Menu  m2=new Menu("Edit");   Menu  m3=new
Menu("Format");
 MenuItem  i=new MenuItem("New"); MenuItem  j=new MenuItem("Open");
MenuItem  k=new MenuItem("Save");  MenuItem  l=new MenuItem("Save as");
   MenuItem a=new MenuItem("Cut");    MenuItem b=new MenuItem("Copy");
MenuItem c=new MenuItem("Delete");    MenuItem d=new MenuItem("Undo");
   MenuItem e=new MenuItem("Word wrap");    MenuItem f=new
MenuItem("Font");
   m.add(m1);    m.add(m2);    m.add(m3);
   m1.add(i);     m1.add(j);    m1.add(k);    m1.add(l);
   m2.add(a);     m2.add(b);    m2.add(c);    m2.add(d);
   m3.add(e);    m3.add(f);
   setMenuBar(m);
   setSize(500,500);
   setVisible(true);
 }
public static void main(String arg[])
 {    TestMenu t=new TestMenu();  }
}
```




## EVENT HANDLING

The actions that have to be performed on the component listening to an event like a mouse clicked on an applet are specified. This is called Event Handling. Once we know which events a particular component supports, we don‘t need to look anything up to react

to that event. Simply take the name of the event class and remove the word Event. Add the word .Listener. to what remains. This is the listener interface we need to implement in the inner class. Implement the interface above and write out the methods for the events we want to capture. For example, we might be looking for mouse movements, so we write code for the mouseMoved( ) method of the MouseMotionListener interface. Create an object of the listener class in step 2. Register it with our component with the method produced by prefixing 'add' to the listener name. Events are signals which are fired when the state of a component is changed (eg: when a button is pressed, when a menu is pressed etc.). In the event of a signal firing it is necessary for us to handle the event based on our requirements. For example we would want to open a new window or close it when a button is pressed, or we would want to list a menu when a menu box is activated (pressed). This paragraph only gives us a gist of what happens when a component is activated. Actually when the internal state of the component is modified a source is generated. This is nothing but a source to the event.

**Listeners**

A single component can take events from different sources. For Example an applet can have sources from the keyboard or from the mouse. So we should make the applet be ready to receive the events from the different sources. This is done by the Listeners which are nothing but interfaces with abstract methods which could be implemented on generation of the corresponding event.

**Drawing strings**

These methods let us draw text strings on the screen. The coordinates refer to the left end of the text's baseline.

**public abstract void drawString (String text, int x, int y)**

The drawString() method draws text on the screen in the current font and color, starting at position (x, y). The starting coordinates specify the left end of the String's baseline.

**public void drawChars (char text[], int offset, int length, int x, int y)**

The drawChars() method creates a String from the char array text starting at text[offset] and continuing for length characters. The newly created String is then drawn on the screen in the current font and color, starting at position (x, y). The starting coordinates specify the left end of the String's baseline.

**public void drawBytes (byte text[], int offset, int length, int x, int y)**

The drawBytes() method creates a String from the byte array text starting at text[offset] and continuing for length characters. This String is then drawn on the screen in the current font and color, starting at position (x, y). The starting coordinates specify the left end of the String's baseline.

**public abstract Font getFont ()**

The getFont() method returns the current Font of the graphics context.

**public abstract void setFont (Font font)**

The setFont() method changes the current Font to font. If font is not available on the current platform, the system chooses a default.

To change the current font to 12 point bold TimesRoman:

**setFont (new Font ("TimesRoman", Font.BOLD, 12));**

**public FontMetrics getFontMetrics ()**

The getFontMetrics() method returns the current FontMetrics object of the graphics context. One should use FontMetrics to reveal sizing properties of the current Font--for example, how wide the "Hello World" string will be in pixels when displayed on the screen.

**public abstract Color getColor ()**

The getColor() method returns the current foreground Color of the Graphics object.

**public abstract void setColor (Color color)**

The setColor() method changes the current drawing color to color.. To change the current color to red, use any of the following:

**setColor (Color.red);**
**setColor (new Color (255, 0, 0));**
**setColor (new Color (0xff0000));**