UNIT-1
History, Introduction and Language, Basics Classes and Objects

Java History:

- Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991. The first version of java is originally called Oak. Developed by James Gosling who is one of the investor of the language.
- After development of the Oak, team was added some new features to the language like multithreading, platform independent, applet etc. But, some reason Sun Microsystems have to change the name of the language.
- And in 1995, it is known as Java. Java was designed for the development of software for consumer electronic devices like TVs, VCRs, toasters and such other electronic machines. This goal had a strong impact on the development team to make the language simple, portable and highly reliable.
- The java team, which included Patrick Naughton, discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. However they modeled their new language Java on C and C++ but removed a number of features in C and C++ that were considered as sources of problems and thus made java a really simple, reliable, portable and powerful language.

Java Features:

- The inventors of Java wanted to design a language that 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. Sun Microsystems officially describes Java with the following attributes:

1. Compiled and Interpreted:

- Generally, in a programming language the source code is first compiled and converted into machine code and then it will be executed. But in Java this thing is different. Java compile first converts the source code into byte code. This byte code is not a machine code. This byte code then Interpreted by java runtime system called Java Virtual Machine (JVM).

2. Platform-Independent and Portable:

- One of the most advantages of Java is its portability. Java program can be easily run on different computer at anytime and anywhere. Java program remains same for any processor, operating systems.
- We can download Java applets from remote computer and run it in over computer. This task is done by Java Virtual Machine which makes it easy to execute.

3. Object Oriented:

- Almost everything in Java is an Object. All program code and data resides within objects and classes. All the object-oriented features like encapsulation, inheritance, and polymorphism are provided in Java. This object-oriented feature make Java program easily understandable and readable.

4. Robust and Secure:

- Java programs are very reliable on any computer system. That is it gives the same output on any system. To better understand how java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional condition. Memory management can be a difficult, tedious task in traditional programming environment.
- For example, in C/C++, the programmer must manually allocate and free the dynamic memory. This sometimes leads to problems, because programmer will either forget to free memory that has been previously allocated or, bad tries to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and de-allocation for you

using garbage collection inbuilt facility.

- Security becomes an important issue for a language that is used for programming on Internet. Threat of viruses and abuse of resources is everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

5. Distributed:

- Java is designed as a distributed language for creating applications on networks. It has the ability to share the data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

6. Multithreaded and Interactive:

- Multithreaded means handling multiple tasks simultaneously. For example, while listening song, one can edit document or scroll the document. Java provides multithreading facility. This means that we need not wait for the application to finish one task before beginning another. This feature greatly improves the interactive performance of graphical applications.

7. High performance:

- Java programs are compiled one time and run one or more time. It takes time only at compilation. After compilation the JVM easily execute this byte code and gives the high performance. Furthermore, multithreading also enhances the performance of execution of java program.

8. Dynamic and Extensible:

- Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods and objects. Because of this it reduces the memory consumptions when the programs are executed.
- Java also supports functions written in other languages such as C and C++. These functions are known as native methods. Native methods are linked dynamically at runtime.

JDK and its components (various tools of JDK):

- JDK stands for Java Developing Kit. As the name suggest JDK is the collection of tools that is used to develop and run the java application. The tools are:

1. applet viewer

- This tool is used to view the applet created by the programmer. Because of this tool it is not necessary to have an java compatible web browser.

2. javac

- This tool is called java compiler. This tool converts source code into byte code. Before executing any program It is necessary to compile it first. This tool produces a class file.

3. java

- This tool is used to execute the java program. It executes class file of java program which is produced by javac. So, it is necessary that before using this tool one has to use javac tool to produce the class file then only the program can be executed.

4. javap

- This tool is called java disassembler, which enables us to convert byte code file into a program file.

5. javah

- This tool is used to create a java header file to use it later in the program.

6. javadoc

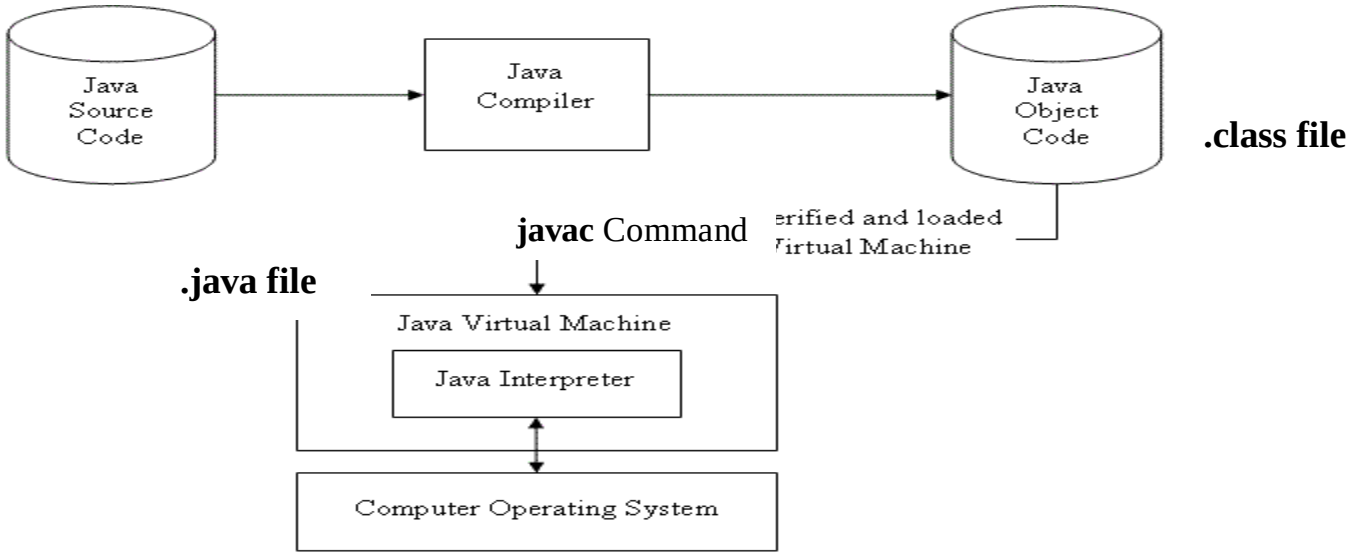
- This tool is used to create HTML-format documentation from java source code file.

7. jdb

- This tool is known as java debugger. It is used to find the error reside in the program. This tool is also useful for the examining the variable of the source code at run time.

Byte code and JVM:

- As with many other programming languages uses a Compiler to convert source code (Human readable) into Executable Programs. Traditional Compilers produces code that can be run by specific hardware only. In contrast, the Java compiler generates architecture-independent ***Byte code***.
- The Byte Codes can only be executed by ***java virtual machine (JVM)***. For that an initialized Java processor chip usually implemented in Software rather then Hardware.
- The key of security and portability problem of java program is solving by Byte code. *Byte code* is the specific set of instruction produced by java compiler.
- To execute Java Byte codes, the virtual machine uses a class loader to fetch the byte code from a disk or from a network.
- Each class file is analyzed by a byte code verifier to ensure that the class is formatted correctly and the class will not corrupt memory when it is executed.
- The Byte code verification phase is performed only once, not continuously as the program runs.
- There are different specifications of JVM for different systems. But it runs the same byte code on different machine. That's why one java program can be executed on any systems.



(1) The source file (.java file) is Compiled by javac compiler and Bytecode is generated if no erros are present. Bytecode is .class file.

(2) This Bytecode is then interpreted by JVM and the output is generated.

Basic structure of Java Program

There may be more than one class in a java program. But you hava to run that class which has the main method.

Basic structure of a java program is as under:

(1) Documentation section (2) Package statement (3) import statements (4) class definition
(5) Main method

(1) Documentation Section :

In this section you can write the documentation for your program like program definition, author information, the logic of your program etc. This optional section. It is written in comment line(//) or (/*…….*/)

(2) Package statement :

If you want to save your program in a package (directory then the package statement is included. This is also an optional section

(3) Import Statement :

If you want to import any library or user-defined package, the import statement is used. Now from this imported package, you can use its classes and methods. This is optional section
Example : import java.util.Date;

(4) Class Definition :

You can define classes in this section. Use class keyword to declare a class. There can be member variables and methods in this class. This is required and it is compulsory section.

(5) Main method :

In your program, you can have more than one class, but there should be one class that contains the main method. This class is run to generate output, because the main method is the entry point for program execution. This is compulsory section.

A First Simple Program:

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

- As we know that before executing the program we have to compile the program. The following command line will compile the java program
  c:\>javac simple.java
- After compilation, one file is created with the same name but with ".class" extension. Now to run the java program, you have to write the following command line.
  c:\>java simple
- As you execute this java program you find the following output on the screen.
  Hello World
- Now let us understand the program. The first line
  class simple {
- declares a class simple. The second line is
  public static void main(String args[]){
- This line begins the main() method. From this line the execution of the program starts.
- The ***public*** is the access specifier which indicates that the main function can be called outside the class. As we know that the java program is executed by JVM which is outsider of the class.
- The keyword ***static*** allows main to be called without instantiating the class. This is necessary since main() is called by the java interpreter before any object are made.
- Keyword ***void*** specifies that the main function does not return any value.
- The String args[] is the parameter of main() function. This parameter holds the arguments passed from command line. Now let us go to the next line

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

- This line output the string "Hello World". Here this first word ***System*** which represents predefined class related to the system. The keyword ***out*** represent the output class which is related to the output stream. println is the function which give the output.

Java IDE(Netbeans and Eclipse)

NetBeans and Eclipse are two most popular IDEs (Integrated Development Editor) in which you can develop, debug and deploy your programs(or you can say projects) easily. These editors provide so many interesting features to develop programs in many languages including java, PHP, C, C++ etc. Both of them are discussed below:

NetBeans:

NetBeans is an open source editor with lots of features to develop any king of complex project. It lets you develop Java desktop, web and mobile applications, while also providing great tools for php and C/C++ developers very quickly and easily. It is totally free and open source with a large community of users and developers around the world.

Following are some striking features of NetBeans IDE :

(1) Fast and Smart Coding:

It's not simply a text editor but much more than that. It helps you to code very easily and quickly by providing automatic line indents, it highlights similar keyword and pair of brackets. Also gives coding tips by opening pop-up menu to suggest the related code.

(2) Easy and efficient Project Management:

It helps you to easily manage your project even if it is getting larger with number of files and files with a large number of lines.

(3) Rapid User Interface Development:

It provides drag and drop facility to rapidly develop and design your forms or pages. you can pick any GUI component from the palette.

(4) Bug Free Code:

The NetBeans provides static analysis tools such as FindBugs tool, for identifying and fixing common errors in java code. Furthermore, the NetBeans Debugger lets you place breakpoints in your source code, add watches, step through your code, run into methods and some other features to debug your program.

You can freely download the latest version of NetBeans from this site:

Netbeans.org/downloads/

Eclipse :

The Eclipse is another very powerful editor for so many languages such as Java, PHP, C, C++ etc. The Eclipse IDE for Developers contains everything that you will need to build java applications. It provides greater java Java editing with validation, incremental compilation, cross-referencing, code assistance, an XML Editor and so many others.

(1) Great debugging support with hyperlink stepping.
(2) A new Quick Access feature to enhance IDE navigation.
(3) Quick Fix/Assit support.
(4) Task-focussed development.

It can be freely downloaded from the like**: http://www.eclipse.org./downloands/**

Language building blocks:

- All java programs are made up of basic elements called tokens. These tokens are identifiers, keywords, literals, white spaces and comments.

1. Identifier

- Identifiers are used to give name to the class, method or variable. An identifier is a group of letters, numbers, underscore and dollar sign characters. They must not begin with number. Java is case sensitive language into identifier.

2. Keywords

- There are 49 reserved keywords currently defined in Java language. This keyword defines a particular meaning. With the use of these keywords, one can define a particular statement. These keywords are reserve. It means that it cannot be used as identifiers. Example of keywords is abstract, boolean, do, if, goto, while etc.

3. Literals

- The constant value in Java is represented using literals. Literal is nothing but a number or a string value. For example

int	101, 0123 (octal) , 0x55A6 (hexa).
long	250l or 250L
double	34.45,
float	30.25f or 30.25F
String	"abc",
char	'a'

4. White spaces

- White space is a basic building block of java program. It separates two tokens in the java program. In java, white space is a space, tab or newline character.

5. Comments

- Comments are used to describe different section of the program. There are three types of comments: 1) single line comment 2) multiple line comment and 3) documentation comment.
- Single line comment contains only one line and it is preceded by // characters. Multiple line comment contains more than one line. This comment starts with /* characters and ends with */ characters. The documentation comment is used to produce an HTML file that documents your program. The documentation comment begins with /** and ends with */.

Primitive data types:

- Java specifies eight simple types of data: byte, short, int, long, char, float, double, and boolean. These data types are classified into four groups:

1. Integer

- Integer data type stores whole numbers, for example, 123,45,62 etc. The size of the values that can be occupied by the variable is depends on its type. There are four type of integers: byte, short, int and long. Their size, range of value is given below:

Type	size	range of value
byte	1 byte	-128 to 127
short	2 bytes	-32768 to 32767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,327,036,854,775,807

- It is recommended that as far as possible, programmer have to use smaller data type to store the value, because smaller data types use less memory and speed up the program execution. For example, If a variable holds a in between 0 to 100 then it is recommended that instead of using int data type, he has to use byte data type.

2. Floating Point Number

- Floating point data type is used to store the value which has fractional parts. For example, 1.2341, 63.231. There are four data types of floating point numbers: float and double which represent single precision number and double precision number, respectively. Their width and approximate range is given below:

Type	Size	Range of value
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308

3. Character

- char data type is used to store character. The standard set of character known as ASCII characters and ISO-Latin-1 are include in this type. It holds one character at a time. This type requires two bytes.

4. Boolean

- Boolean type is mainly used to store one of the values: true or false. Variables with this data type are used in conditional statements to take decision.

Type Conversion and Casting:

- In a programming environment it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible than type conversion takes place. This conversion is done automatically by Java. For example, an integer value is automatically assigned to a long variable. It is not necessary that this type of automatic type conversion take place. There are two rules to convert a value from one type to another type.

i. The two types are compatible.
ii. The destination type is larger than the source type.

- The first point says that the source and destination should be compatible. It means that if one is in numeric form than another one must be in numeric form. It is not possible to assign a numerical value to a boolean variable.
- The second point says that the source should be smaller that the destination variable in type. For example it is possible that a integer type value is automatically converted into byte.
- There are occasions where the automatic type conversion does not take place. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because byte is smaller than int. This type of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

  (target-type) value

- Here target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the Integer value is larger than the range of a byte.

```
int a=128;
byte b;
b=(byte)a;
```

- Here variable b will contain 1 as the calculation of 128%127.

Operators and its Precedence and Associativity:

- Java supports a rich set of operators. Operators are specific symbols that performs specific task. These Operators are classified into following categories:

1. Arithmetic Operator:

- Java provides all the arithmetic operators. These operators are +(plus), -(minus), *(multiplication), /(division), %(Modulo division). All work same as in another languages. Each of these operators has two operands. + and – operator makes the operand positive or negative, when they are used with only one operand.

2. Relational Operator:

- We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators. All relational operator and their meaning are given below:

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	is not equal to

- Relational operator always produce Boolean values. For example, if x=10, then X<11 will produce true.

3. Logical operator:

- Logical operator is used to combine the result of two or more relational expression. Java provides following logical operator:

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

- The logical operator is surrounded by relation expressions. The value generated by the logical expression is base on the operator and its operand. The logical expression generates true or false value according to the truth table given below:

op-1	op-2	op-1 && op-2	op-1 || op-2	!op-1
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

4. Assignment Operator:

- Assignment operator assigns value of an expression or constant to a variable.

```
x=1;
y=x+1;
```

- Here variable y will assign the value 2. Java has a set of shorthand operator, which takes the following form.

```
Variable op= exp;
```

- **op** means any arithmetic operator. Let us use the shorthand operator in the expression y = y+1.

```
Y += 1;
```

5. Increment and Decrement Operator:

- Java provides very useful operator called increment and decrement operator. ++ is the increment operator and - - is decrement operator. Both are unary operator. The ++ operator add 1 to the

operand while - - subtracts. Both operators are used in the following form:

```
++m or m++;
--m or m--;
```

- While ++m and m++ mean the same thing when they form statements independently, they behave differenently when are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

- In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as:

```
m = 5;
y = m++;
```

- Then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increment the operand.

6. Conditional Operator:

- The character pair ? : is known as conditional or ternary operator. This operator is used to construct conditional expression of the form

```
exp1?exp2:exp3
```

- where, exp1,exp2 and exp3 are expression.
- The operator ? : works as follows: exp1 is first evaluated. If it is true then the exp2 will be evaluated and becomes the value of the conditional expression. If it is false then the exp3 will be evaluated and becomes the value of the conditional expression.

```
a=10;
b=11;
x=(a>b)?a:b;
```

- Here x assigns the value of b.

7. Bitwise Operator

- Bitwise operators are use with data at bit level. This operator is mainly use for testing bits or shifting left to right or right to left. Different bitwise operator are given below:

Operator	**Meaning**
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right
>>>	shift right with zero fill

- **precedence and associativity:**
  - In java, the precedence of an operator decides in which way the expression should be evaluated. Each operation has been given precedence. There are distinct levels of precedence. The operators at the higher level are evaluated first. The operator with the same precedence is evaluated either from left to right or right to left according to their associativity. Following table provides the complete list of operator with its associativity and precedence in the form of rank.i.e. rank 1 means higher precedence.

Operator	**Associativity**	**Rank (Precedence)**
.(dot) , (), []	Left to right	1
- (Unary), ++, --, !, ~	Right to left	2

*, /, %,	Left to right	3
+, -	Left to right	4
<<,>>,>>>	Left to right	5
<,<=,>,>=	Left to right	6
==,!=,	Left to right	7
&	Left to right	8
^	Left to right	9
|	Left to right	10
&&	Left to right	11
||	Left to right	12
?:	Right to left	13
=,op=	Right to left	14

- It is very important to note carefully, the order of precedence and the associativity of operators. Consider the following statement:

  If(x==10+15 && y<10)

- Here, + has higher precedence so at first, the addition will take place. Then after the relational operator < has higher precedence. So that expression will be evaluated and gives true or false value then the expression will == operator will evaluate according to its precedence and gives true or false value. At last the operator && will be evaluated on the value of the last two value of the expression "x==25" and "y<10".

Flow Controls:

- Programming languages have control statements, which control the flow of execution.

1. Condition statements:

- Condition statements are those statements, which executes one of the blocks of statement.

a) If-else statement:

- The **if** statement branches the execution of the java program. It specifies the route to selection based on a specific condition. Here is the general form of **if** statement:

```
if(condition)
{
        statement block1;
}
else
{
        statement block2;
}
```

- Here each block contains one or more statements. The condition statement is any expression that returns boolean value. The **if** statement works like this: if the condition statement returns true then the statement block1 will be executed. If it returns false than the statement block2 will be executed. Let us understand **if** statement with example:

```
int a, b;
if(a>b)
        a=0;
else
        b=0;
```

- Here, the condition "a>b" is true than the value of "a" will be 0 otherwise value of the "b" will be 0.

b) Nested if:

- When one if statement occurs in another if statements block then it is called **nested if**.

```
int a,b,c,min;
if(a>b)
{
        if(b>c)
                min=c;
        else
                min=b;
}
else
{
        if(a>c)
                 min=c;
        else
                min=b;
}
```

- In this portion of the program, if statements are nested to find the minimum value among the three values. First the "a>b" condition is checked. If it is true then the second condition "b>c" will be checked. If it is true then min variable will be assign the value of "c" variable. If it is false than the minimum value will be “b” value.
- If the "a>b" condition is false than the "a>c" condition of else part will be evaluated. If it is true than the value of “c” variable will be assign to min variable otherwise the value of "b" variable will be assign to min variable.

c) The if-else-if Ladder:

- When there are more than two branches at that time we have to use multiple if and else if statements. The general form of if-else-if ladder statement is:

```
if(condition1)
        statement block1;
else if(condition2)
        statement block2;
.
.
else
        statement block n;
```

- Here in this form, condition1 is evaluated first, if it is true then the statement block 1 will be executed. If it is false then condition2 will be evaluated, if it is true then the statement block 2 will be executed. This will happen until one condition will be true. If all the conditions are false then the else part will be executed.

d) **Switch:**

- The switch statements java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statement. Here is the general form of switch statement:

```
switch(expression)
{
        case value1:
```

```
                statement block1;
                break;
        case value2:
                statement block2;
                break;
        .
        .
        case valueN:
                statement blockN;
                break;
        default:
                default statement block;
}
```

- In switch statement, the expression gives one value which is compared with the values written with case statements. If one case is evaluated true then the statement block related to that case would be executed. Following portion of the program give the example of switch statement.

```
int i;
switch(i)
{
        case 1:
                System.out.println("One");
                break;
        case 2:
                System.out.println("Two");
                break;
        case 3:
                System.out.println("Three");
                break;
}
```

- Here this program will print the number in word format.

2. Looping Statement:

- Loop statements are those statements, which executes a block of statements for a specific number of times. There is a condition, which controls the execution of loop. There are three types of Loop statements: 1) While 2) do 3) for

a) While:

- While loop is Java's fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition)
{
        body of loop
}
```

- The condition can be any boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

```
i=0;
while(i<10)
{
        System.out.println("Index is " + i);
        i=i+1;
}
```

- The statements inside the while block will repeated 10 times and in that response the statement with index number will be printed 10 times.

b) Do:

- In while loop the conditional expression is evaluated first. If it is true than the block of statements will be executed otherwise the block will be skipped. In **do** loop the placement of conditional expression is evaluated at the end of the block. Let us see the structure of do-while statement.

```
do
{
        block of statements
}while(condition);
```

- As shown in the syntax, the block of statements is executed first. After this execution the condition will be checked. If this condition is true then and then only the block of statement inside the curly brace will be executed. Otherwise the next statement after the while statement will be executed.

c) For:

- The for loop is very powerful construct. Here is the general form of for statement:

```
for( initialization; condition; iteration)
{
        block of statements;
}
```

- The initialization part of this construct will initialize the variable. The condition statement can be any expression that produces boolean value. The iteration part may contain increment statement or decrement statement.
- When the control enters into the loop the initialization part evaluated first. Then the condition will be checked. If it is true then the block of statements inside the curly braces will be executed. In the next iteration of the loop the iteration part will be executed first. After that the condition will be checked. This procedure continue until the condition is false. Let us take an example:

```
For(i=0;i<10;i++)
{
        System.out.println("value of I is" + i);
}
```

- This portion of the program will print 10 lines with the increasing value of i.

d) Break:

- By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When break statement encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is an example:

```
for(int i=0;i<100;i++)
{
        if(i==10)
                break;
        System.out.println("i : "+ i);
}
System.out.println("Loop complete");
```

- Here, for loop should iterate 100 times but the **break** statement inside the **if** statement will break the loop after 11 times of the iteration.
- Note that if loops are nested then break will terminate innermost loop.

e) Continue:

- Sometimes you want to skip some part of the loop on some condition for a particular iteration of the loop. The **continue** statement performs such an action. In while and do- while loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a for loop, control first goes to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. Let us take an example:

```
for(int i=0;i<10;i++)
{
        System.out.print(i + " ");
        if(i%2 ==0)
                continue;
        System.out.println("");
}
```

- This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output of the program:

```
0 1
2 3
4 5
6 7
8 9
```

Arrays:

- Array is a collection of values of same type, which are referred by same name. Each value is referred by a number, which is called index number. Arrays of any type can be created and may have one or more dimensions. Each subscript starts with 0.

1. One Dimension Array

- One dimension array has one index number through which an element of the array can be identified. One dimension array can be declare as following:

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

- This statement will create an array of five elements. If you want to store value 10 into 4^{th} element then you can access 4^{th} element with 3 subscript numbers.

```
a[3]=10;
```

- You can also use these array elements with its subscript number in any mathematical expression.

2. Multidimensional Array

- An array which has more than one dimension then it is called multi-dimension array. The two dimension array is used to store values of a table. First dimension indicates the row index while second dimension indicates column index. The declaration syntax is as given below:

```
Datatype arr[][]=new datatype[totalrows][totalcolumns];
```

- Here this type of array contains (totalrows * totalcolumns) elements. Now lets take an example:

```
int a[][]=new int[3][2];
```

- This array contains 3 rows and 2 columns. Any element can be identified by two subscript number. You can also create three-dimension array. For example, lets declare three-dimension array.

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

3. Jagged Array

Jagged Array is a special type of multi-dimensional array in which each row has different number of columns. Therefore it is known as variable length array

To declare a Jagged Array, you have to specify only row size and leave the columns size blank.

Following is the syntax to declare a Jagged Array.

Syntax

```
Data_type arr_name[ ][ ]=new data_type[Rowsize][ ];
```

For example :

```
Double d[ ][ ]=new double[5][ ];
Int J[ ][ ]=new int[4][ ];
```

Here, J is a jagged array having 4 rows. Now you can declare the size of columns for each row as shown below :

```
J[0]=new int[3];
J[1]=new int[2];
J[2]=new int[4];
J[3]=new int[3];
```

J[0][0]	J[0][1]	J[0][2]	
J[1][0]	J[1][1]		
J[2][0]	J[2][1]	J[2][2]	J[2][3]
J[3][0]	J[3][1]	J[3][2]	

First row i.e.J[0] contains 3 columns, second row (J[1]) has 2 columns, third row (J[2]) has 4 columns and so on.

Following is an example of Jagged Array in which we havae initialized all the rows of array.

Jagged Array Example :

//simple program of jagged array…..

```
class jaggedarryex
{
        public static void main(String args[])
        {
                Int a[ ][ ]={
```

```
                {1,2,3,4},
                {1.2),
                {1,2,3},
                {1,2,3,4,5}
                };
        System.out.println("\n Jagged Array…….\n\n");
        for(int i=0;i<a.length; i++)
        {
                for (int j=0; j<a.length; j++)
                {
                        System.out.print(a[i][j]+" ");
                }
                        System.out.print("\n");
        }
    }
}
```

Command-line arguments

You can pass arguments to the main() method. This can be done at runtime. As we know that the main() method takes array of string as argument, the arguments given to the main() method will be assigned to this array. Thus giving command-line argument is quite easy in java. Following program shows how to give and receive command-line arguments in a program:

```
//passing arguments to main() method…
class cmdlineex
{
        public static void main(String arg[])
        {
                for(int i=0; i<arg.length ; i++)
                {
                        System.out.println("argument "+i +" : "+arg[i]);
                }
}
```

Output :

```
D:\jitu>javac cmdlineex.java
D:\jitu>java cmdlineex 123 abc hello 12.34
Argument 0 : 123
Argument 1 : abc
Argument 2 : hello
Argument 3 : 12.34
```

Introduction to Classes:

The class is the core of the object oriented language java. Any concept you wish to implement in a java program must be encapsulated within a class. The most important things to understand about a class is that it defines a new data type using this new type can be create object (variable) of that type. Thus, Class is like a template for creating of its object. We can define the class as follow.

```
                class  ClassName
                {
                        type variable_1;
                        type variable_2;
                        :
                        type variable_N;
```

```
        type methodName1(parameter_list)
        {
                //Body of the method1
        }
        type methodName2(parameter_list)
        {
                //Body of the method2
        }
        :
        type methodNameN(parameter_list)
        {
                //Body of the methodN
        }
}
```

- Following is the code for a class called SimplePoint that represents a point in 2D space:

```
class SimplePoint
{
        int x ;
        int y;
        public void set()
        {
                x=0;
                y=0;
        }

}
```

- This segment of code declares a class or a new data type called SimplePoint. The SimplePoint class contains two integer member variables x, y and one method set(). Here method set() will set the value of the variable x, y.

Declaring Objects:

When you create a class, you are creating a new data type. You can use this type to create a variable of that class type. This variable does not an object. It is simply a variable that can refer to an object of that class type or simply reference variable of that class type.

If you want to create an object then you have to acquire physical memory of the object and assign it to that reference variable. You can do this using the **new** operator. The **new** operator allocates memory for an object and returns a reference to it.

When you create a new SimplePoint object, space is allocated for the object and its members x and y. Note that the space for method will be allocated only once at the time of declaration of the class. Now let us create an object of class SimplePoint.

```
SimplePoint sp;  // Declare reference to object.
sp = new SimplePoint(); // Allocate a object to reference.
```

Here first sp is the reference variable of the class and it is refer the object of the class that is return by the new operator. We can also define the object like below.

```
SimplePoint sp = new SimplePoint();
```

Here both statements are work together to create an object of the class type.

You can access the members outside the class with the use of "."(dot) operator with the name of the object. For example,

```
sp.x=0;
```

You can also access the method of the class with the "." operator like below.

```
sp.set();
```

You can define all type of the method into the class which is allow into C++ / C.

You can also define method overloading as like C++ and also define variable argument method, which is taking variable argument from the calling time.

Overloading methods:

In java it is possible to define two or more methods within the same class that share the same name and parameter declaration is different. When this is the case, the method is said to be overloaded, and the process is referred to as method overloading.

When an overloaded method is invoked, java uses the type and number of parameters as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods may have differed in the type and number of their parameters.

When java encounters a call to an overloaded method, it simply executes the version of a method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading.

```
        class  OverLoad
{
        void test()
        {  System.out.println("No parameter");  }

        void test( int a)
        {  System.out.println("A =  " + a);  }

        void test( int a , int b)
        {  System.out.println("A and B is : " + a + "  " + b );  }
}

class OverLoadDemo
{
        public static void main(String s[])
        {
                OverLoad obj = new OverLoad();
                // call all version one by one
                obj.test();
                obj.test(10);
                obj.test(10,20);
        }
}
```

This program generates following Output.

```
No parameter
A = 10
A and B is : 10  20
```

In this program test() method is overload three times. The first version is without parameter, the second takes one parameter and thirds takes two parameters.

As like method you also overload the constructor of the any class to initialize object. Constructor overloading is work as like as method overloading.

In java, constructor overloading is also define as like C++ language.

Constructors:

Basic requirement for initialization is so common, Java allows object to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes objects when they created, before the new operator completes. Constructor is as like as the method but only different that constructor name is same as the class name and they have no return type, not even void. This is because the implicit return type of constructor is implicitly class type itself.

You can also define more then one constructor into one class it is known as the constructor overloading as like method overloading. You are calling the constructor explicit as like the C++ language. You can define the constructor as like below.

```
class SimplePoint
{
    int x ;
    int y;
    SimplePoint()
    {
            x=10;
            y=10;
    }
}
```

Here whenever the class SimplePoint is instantiated, the constructor (method) SimplePoint() will be called automatically which will initialize the value of the member variable x with 10 and y with 10.

If you want to pass the parameter to the constructor then also it is possible.

```
SimplePoint(int i1, int i2)
{
        x = i1;
        y = i2;
}
```

Here i1 and i2 is the parameter of the constructor when we create object then we have to pass two parameter of int type to call parameterized constructor.

```
SimplePoint s1 = new SimplePoint(5,7);
```

When s1 object is created then parameterized constructor is called in that 5 and 7 are passed to the parameter i1 and i2 respectively and 5 is assign to x and 7 is assign to y.

Use of 'this' Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the 'this' keyword. 'this' can be used inside any method to refer current object.

That is, 'this' is always a reference to the object on which the method was invoked. You can use 'this' any where a reference to an object of the current class type is permitted.

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

The use of 'this' is redundant, but perfectly correct. Inside Box( ) 'this' will always refer to the invoking object.

Garbage Collection and finalize() method

Garbage collection Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.

Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs

The finalize () Method Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize() method. The java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize() method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the java run time calls the finalize() method on the object.

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. It is important to understand that finalize() is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize() for normal program operation..

Native,volatile and transient keywords

Native :

Java supports native methods. Native methods are those methods which are written in some another language like c or c++. Native keyword is used to declare a method as native. A simple statement to call a native method is:

```
public native returnType methodName();
```

volatile

-- The volatile keyword is mostly used to indicate that a member variable of a class may get modified asynchronously by more than one thread. -- This thing is noticable that the volatile keyword is not implemented in many Java Virtual Machines. -- The volatile keyword from the side of compiler tries to guarantee that all the threads should see the same value of a specified variable

Transient

- The transient keyword is applicable to the member variables of a class. -- The transient keyword is used to indicate that the member variable should not be serialized when the class instance containing that transient variable is needed to be serialized.

Inheritance Basics:

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. To see how, let's begin with a short example. The following program creates a superclass called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.

```
// A simple example of inheritance.
// Create a superclass.
class A
{
        int i, j;
        void showij()
        {
        System.out.println("i and j: " + i + " " + j);
        }
}
// Create a subclass by extending class A.
class B extends A
{
        int k;
        void showk()
        {
        System.out.println("k: " + k);
        }
        void sum()
        {
        System.out.println("i+j+k: " + (i+j+k));
        }
}
class SimpleInheritance
{
        public static void main(String args[])
        {
        A superOb = new A();
        B subOb = new B();
        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        /* subclass has access to all public members of its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
        }
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
```

```
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

When Constructors are called:

In the inheritance the default constructor is calling the first base class and then calls the derived class constructor. Here simple example of the constructor calling in the inheritance.

```
class A
{
        int a;
        A()
        {
                System.out.println("Constructor of Base ");
                a = 0;
        }
}

class B extends A
{
        int b;
        B()
        {
                System.out.println("Constructor of Derived ");
                b=0;
        }
        void disp()
        {
                System.out.println("A from Base: "+a);
                System.out.println("B from Derived: "+b);

        }
}
class constMain
{
        public static void main(String s[])
        {
                B  b1 = new B();
                b1.disp();
        }
}
```

This program will generate following output:

```
Constructor of Base
Constructor of Derived
A from Base: 0
B from Derived: 0
```

As you can see the constructor is called in order of derivation. Here when the object of the derived class is generated then the constructor of derived class is called from that constructor it will check that in the base class there are default constructor is available or not. If there are constructor is available in the base class then it will call first and then execute the derived class constructor.

VarArgs

In general, your method definition specifies the number of arguments you have to pass. But the VarArgs is a concept in which you can define method that takes variable number of arguments. It means that you can pass as many numbers of arguments to a function as you want with no restrictions to match method's type signature.

To declare VarArgs method, following syntax is used :

```
Access-specifier   Return_Type    Method_name (Data_type... argName)
{
        //Method Body
}
```

In method argument, you have to specify 3 dots (Ellipses) as shown in the syntax.

Now at the time of calling this method, you can pass any number of arguments to this function.

Example

```
//simple Example of VarArgs…
class VarArgsex
{
        public static void test(String…args)
        {
                System.out.print("Arguments:  "+args.length +"→");
                for(String s: args)
                {
                        System.out.print(s+" ");
                }
                System.out.println();
        }
        Public static void main(String args[ ])
        {
                test("Hi","hello");
                test ("welcome");
                test ("good","bye","thank","you");
        }}
```

UNIT-2

Inheritance, Java Packages

Types of Inheritance in Java

There are three types of inheritance in java:

(1) Single Inheritance :

In single inheritance there is one super class and one subclass. The above examples are of single inheritance.

(2) Hierarchical Inheritance

In this type, there is one super class and more than one subclass of it. Its general form is:

```
class  A{
//statements
}
class B extends A { //B's super class is A
//statements
}
class C extends A{ //C's super class is also A
//statements
}
```

(3) Multilevel Inheritance

In this type one class extends super class and another subclass extends this subclass.

General form :

```
class  A{
//statements
}
class B extends A{
//statements
}
class C extends B{
//statements….
}
```

Creating a Multilevel Hierarchy:

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A,B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass BoxWeight is used as a superclass to create the subclass called Shipment.Shipment inherits all of the traits of BoxWeight and Box, and adds a field called cost, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.
// Start with Box.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
```

```
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1;  // use -1 to indicate
height = -1; // an uninitialized
depth = -1;  // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Add weight.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
```

```
}
}
// Add shipping costs
class Shipment extends BoxWeight {
double cost;
// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}
// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}
// default constructor
Shipment() {
super();
cost = -1;
}
// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);
cost = c;
}
}
class DemoShipment {
public static void main(String args[]) {
Shipment shipment1 =new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =new Shipment(2, 3, 4, 0.76, 1.28);
double vol;vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();
vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}
}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
```

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

Because of inheritance, Shipment can make use of the previously defined classes of Box and BoxWeight, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: super( ) always refers to the constructor in the closest superclass. The super( ) in Shipment calls the constructor inBoxWeight. The super( ) inBoxWeight calls the constructor in Box. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters "up the line." This is true whether or not a subclass needs parameters of its own.

Method Overriding:

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

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

k: 3

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A.

If you wish to access the superclass version of an overridden function, you can do so by using super. For example, in this version of B, the superclass version of show( ) is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}

void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

If you substitute this version of A into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, super.show( ) calls the superclass version of show( ). Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not
// overridden.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// overload show()
void show(String msg) {
```

```
System.out.println(msg + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
}
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

The version of show( ) in B takes a string parameter. This makes its type signature different from the one in A, which takes no parameters. Therefore, no overriding (or name hiding) takes place.

Using final to Prevent Overriding:

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

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

System.out.println("Illegal!");
}
}
```

Because meth( ) is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result. Methods declared as final can sometimes provide a performance enhancement:The compiler is free to inline calls to them because it"knows"they will not be overridden by A subclass.When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.Inlining is only an option with final methods.Normally, Java resolves calls to methods dynamically, at run time.This is called latebinding. However, since final methods cannot be overridden, a call to one can be resolved at compile time.This is called earlybinding.

Dynamic Method Dispatch:

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

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

The output from the program is shown here:

Inside A's callme method
Inside B's callme method
InsideC'scallmemethod

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A,B,and C are declared. Also, a reference of typeA, called r,is declared.

The program then assigns a reference to each type of object to r and uses that reference to Invoke callme().As the output shows, the version of callme() execute d is determined by The type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme( ) method.

Using Abstract Classes:

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class Figure used in the preceding example. The definition of area( ) is simply a placeholder. It will not compute and display the area of any type of object. As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods which mustbe overridden by the subclass in order for the subclass to have any meaning. Consider the class Triangle. It has no meaning if area( ) is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method. You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

abstract type name(parameter-list);

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the classkeyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Here is a simple example of a class with an abstract method, followed by a class
which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
```

```
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class A implements a concrete method called callmetoo( ). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit. Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example. Using an abstract class, you can improve the Figure class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares area( ) as abstract inside Figure. This, of course, means that all classes derived from Figure must override area( ).

```
// Using abstract methods and classes.
abstract class Figure {
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
```

```
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);

Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

As the comment inside main( ) indicates, it is no longer possible to declare objects of type Figure, since it is now abstract. And, all subclasses of Figure must override area( ). To prove this to yourself, try creating a subclass that does not override area( ). You will receive a compile-time error.

Although it is not possible to create an object of type Figure, you can create a reference variable of type Figure. The variable figref is declared as a reference to Figure, which means that it can be used to refer to an object of any class derived from Figure. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Interfaces:

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Defining an Interface:

An interface is defined much like a class. This is the general form of an interface:

```
accessinterface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
```

```
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Here, accessis either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public. Here is an example of an interface definition. It declares a simple interface which contains one method called callback( ) that takes a single integer parameter.

```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces:

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
accessclassclassname [extends superclass]
[implements interface [,interface...]] {
// class-body
}
```

Here, accessis either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition. Here is a small example class that implements the Callback interface shown earlier.

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that callback( ) is declared using the public access specifier. When you implement an interface method, it must be declared as public.It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```
class Client implements Callback {
// Implement Callback's interface
```

```
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +"may also define other members 2”);
}
}
```

Applying Interfaces (Using Interface we can apply multiple inheritance):

To understand the power of interfaces, let’s look at a more practical example. In earlier chapters you developed a class called Stack that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be “growable.” The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods push( ) and pop( ) define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let’s look at two examples. First, here is the interface that defines an integer stack. Put this in a file called
IntStack.java. This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop(); // retrieve an item
}
```

The following program creates a class called FixedStack that implements a
fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
FixedStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
if(tos==stck.length-1) // use length member
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
```

```
else
return stck[tos--];
}
}
class IFTest {
public static void main(String args[]) {
FixedStack mystack1 = new FixedStack(5);
FixedStack mystack2 = new FixedStack(8);
// push some numbers onto the stack
for(int i=0; i<5; i++) mystack1.push(i);
for(int i=0; i<8; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
System.out.println(mystack2.pop());
}
}
```

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// Implement a "growable" stack.
class DynStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
// if stack is full, allocate a larger stack
if(tos==stck.length-1) {
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++) temp[i] = stck[i];

stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
```

```
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest2 {
public static void main(String args[]) {
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);
for(int i=0; i<20; i++) mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```

The following class uses both the FixedStack and DynStack implementations. It does so through an interface reference. This means that calls to push( ) and pop( ) are resolved at run time rather than at compile time. access stacks through it.

```
*/
class IFTest3 {
public static void main(String args[]) {
IntStack mystack; // create an interface reference variable
DynStack ds = new DynStack(5);
FixedStack fs = new FixedStack(8);
mystack = ds; // load dynamic stack
// push some numbers onto the stack
for(int i=0; i<12; i++) mystack.push(i);
mystack = fs; // load fixed stack
for(int i=0; i<8; i++) mystack.push(i);
mystack = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
System.out.println(mystack.pop());
mystack = fs;
System.out.println("Values in fixed stack:");
for(int i=0; i<8; i++)
System.out.println(mystack.pop());
}
}
```

In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push( ) and pop( ) defined by the DynStack implementation. When it refers to fs, it uses the versions of push( ) and pop( ) defined by FixedStack. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Interfaces Can Be Extended:

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();

ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

As an experiment you might want to try removing the implementation for meth1( )inMyClass. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces. Although the examples we've included in this book do not make frequent use ofpackages or interfaces, both of these tools are an important part of the Javaprogramming
environment.Virtually all real programs and applets that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

Interfaces

An interface describes fields and methods, but does not implement them. An interface-declaration may contain field descriptions, method descriptions, class declarations, and interface declarations.

The declarations in an interface may appear in any order:

interface-modifiers interface I extends-clause {
field-descriptions
method-descriptions
class-declarations
interface-declarations
}

An interface may be declared at top-level or inside a class or interface, but not inside a method or constructor or initializer. At top-level, the interface-modifiers may be public, or absent. A public interface is accessible also outside its package. The extends keyword may be absent or have the form

extends I1, I2, ...

where I1, I2, . . . are interface names. If the extends-clause is present, then interface I describes all those members described by I1, I2, . . . , and interface I is a subinterface (and hence subtype) of I1, I2, ... Interface I can describe additional fields and methods, but cannot override inherited member descriptions.

A field-description in an interface declares a named constant, and must have the form

field-desc-modifiers type f = initializer;

where field-desc-modifiers is a list of static, final, and public; all of which are understood and need not be given explicitly. The field initializer must be an expression involving only literals and operators, and static members of classes and interfaces.

A method-description for method m must have the form:

method-desc-modifiers returntype m(formal-list) throws-clause;

where method-desc-modifiers is a list of abstract and public, both of which are understood and need not be given explicitly.

A class-declaration inside an interface is always implicitly static and public.

- Classes implementing interfaces

A class C may be declared to implement one or more interfaces by an implements keyword:

class C implements I1, I2, ...{
classbody
}

In this case, C is a subtype of I1, I2, and so on. The compiler will check that C declares all the methods described by I1, I2, . . . , with exactly the prescribed signatures and return types. A class may implement any number of interfaces. Fields, classes, and interfaces declared in I1, I2, . . . can be used in class C.

Here three interface declared

interface Colored { Color getColor(); }

Now class implementing this interfaces. Note that the methods getColor must be public because they are implicitly public in the above interfaces.

```
class ColoredPoint implements Colored {
Color c;
ColoredPoint(Color c) { this.c = c; }

public Color getColor() { return c; }
}
```

overloading and overriding of methods

In a class, if two methods have the same name and a different signature,it is known as overloading in Object oriented concepts.

For example, take the case of a Shape Class where you have a method with the name DrawShape();
This method has two definitins with different parameters.

```
public void DrawShape(int x1, int y1,int x2,int y2)
{
// draw a rectangle.
}
public void DrawShape(int x1,int y1)
{
// draw a line.
}
```

These two methods do different operations based on the parameters. Through the same interface (method name) the user will be able to do multiple tasks (draw a line and rectangle). This is called overloading, and it is an example of polymorphism.

Overriding means, to give a specific definition by the derived class for a method implemented in the base class.

For example:

```
Class Rectangle
{
publc void DrawRectangle()
{
// this method will draw a rectangle.
}
}
Class RoundRectangle extends Rectanlge
{
public void DrawRectangle()
{
//Here the DrawRectangle() method is overridden in the
// derived class to draw a specific implementation to the
//derived class, i.e to draw a rectangle with rounded corner.
}
}
```

In the above example, the RoundedRectangle class needs to have its own implementation of the DrawRectangle() method,i.e to draw a rectangle with rounded corners, so it overloaded and gave it implementation.

static members

In place of global variables as in C/C++, Java allows variables in a class to be declared static:

```
public class A
{
  static int k_var;
 }
```

A single memory location is assigned to this variable and it exists and is accessible even when no objects of class are created. These static variables are also called class variables, since they belong to the class as a whole.

Similarly, methods can also be declared as static, and that methods are called static methods or class methods

```
 public class A {
  static void A_method(float x){ }
```

}

These class methods also can be called even when no instance of the class exists. The code in a class method can only refer to static variables and the argument variables. This static method can be access with the name of the class and dot operator.

final variable and methods

The final keyword is used in several different contexts as a modifier meaning that what it modifies cannot be changed in some sense. You may declare member to be final. When a member is declared final, it is a constant which will not and cannot change. It can be set once (for instance when the object is constructed, but it cannot be changed after that.) Attempts to change it will generate either a compile-time error or an exception (depending on how sneaky the attempt is).

Fields that are both final, static, and public are effectively named constants. For instance a physics program might define member variable "c" as the constant, the speed of light as

```
public class Physics {
 public static final double c = 2.998E8;
 }
```

You can also declare that methods are final. A method that is declared final cannot be overridden in a subclass. The syntax is simple, just put the keyword final after the access specifier and before the return type like this:

```
public final String convertCurrency()
```

abstract methods and class

The abstract keyword can be used on classes and methods. A class declared with the "abstract" keyword cannot be instantiated, and that is the only thing the "abstract" keyword does. Example of declaring a abstract class:

```
abstract CalendarSystem (String name);
```

When a class is declared abstract, then, its methods may also be declared abstract.

When a method is declared abstract, the method can not have a definition. This is the only effect the abstract keyword has on method. Here's a example of a abstract method:

```
abstract int get_person_id (String name);
```

Abstract classes and abstract methods are like skeletons. It defines a structure, without any implementation.

Note that, only abstract classes can have abstract methods. Abstract class does not necessarily require its methods to be all abstract.

Classes declared with the abstract keyword are solely for the purpose of extension (inheritance) by other classes.

The following is a code template to experiment with. Begin by commenting out lines involving H2. Experiment with just defining a abstract class. See what is allowed, what is not. Then, define H2 that extends a abstract class. Get a feel of what needs to be there or not. Or, what kind of complaints compiler makes. Finally, create a object of a class that is inherited from a abstract class.

```
// a abstract class
abstract class H {
   int x;

   /* abstract */ int y; // variables cannot be abstract
   /* abstract */ H () {x=1;} // constructer cannot be abstract
   void triple  (int n) {x=n*3;}; // a normal method
```

```
   static int triple2 (int n) {return n*3;}; // a static method in abstract class is ok.
   abstract void triple3 (); // abstract method. Note: no definition.

   // static abstract void triple3 (int n); // abstract static cannot be combined
   int returnMe () {return x;}
}
// H1 extends (inherits) H. When a class extends a abstract class,
// it is said to "implement" the abstract class.
class H1 extends H {
   void triple3 () {x=x*3+1;} // must be defined, else compiler makes a complaint.
   //Also, all return type and parameter must agree with the parent class.
}
public class abbst {
   public static void main(String[] args) {
      // H xx = new H(); // abstract class cannot be instantiated
      H1 myO = new H1(); // abstract class cannot be instantiated
      myO.triple3();
      System.out.println(myO.returnMe());
   }
}
```

The Purpose Of Abstract Classes

Abstract class and methods are complications arising out of the pure Object Oriented Paradigm. Its purpose is to serve purely as a umbrella of classes. That is to say: abstract class is a parent for a collection of sub classes, where itself declares certain methods but doesn't define any implementations. This is so that, sub classes all inherit the same set of methods and variables, and are free to implement them or add in addition.

visibility controls and modifiers:

Access or visibility rules determine whether a method or a data variable can be accessed by another method in another class or subclass.

We have used the public modifier frequently in class definitions. It makes classes, methods, or data available to any other method in any other class or subclass. The modifiers that are used into protection of the data members of the class is called access modifiers. These are public, private, default, protected. Generally modifiers takes place in the declaration of the class member. For example, in create a member variable of type public, we can use the following form.

public int x;

Access Modifiers

Java provides for 4 access modifiers :

1. **public** – The class member declared with this modifier can be access by any other class anywhere.
2. **protected** - The class member declared with this modifier is accessible by the package classes and any subclasses that are in other packages .
3. **Default** – It is also known as "package private". The class member declared with this modifier can be accessible to classes in the same package but not by classes in other packages, even if these are subclasses.
4. **private** - The class member declared with this modifier is accessible only within the class. Even methods in subclasses in the same package do not have access.

Note that a java file can have only one public class.

These access rules allow one to control the degree of encapsulation of your classes. For example, you may distribute your class files to other users who can make subclasses of them. By making some of

your data and methods private, you can prevent the subclass code from interfering with the internal workings of your classes.

Also, if you distribute new versions of your classes at a later point, you can change or eliminate these private attributes and methods without worrying that these changes will affect the subclasses. The following table describes the accessibility of different modifier.

	Private	Default	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package subclass	No	Yes	Yes	Yes
Same Package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Other Modifiers:

final, abstract, static, synchronized, native, volatile, transient.

final:

This modifier when used with a member variable, then that member variable becomes constant for that object. For example,

final int x=10;

Here this statement will create x as constant whose value will not be changed till the object exists. When it is used with any method then that method can not be overridden by the subclass.

abstract:

Whenever you use this modifier with the class declaration, then that class can not be instantiated. It can only be used as base class for other class. When it is used in declaring the method then that method can not have the body. It must be overridden by it derived class.

static:

There may be times in your program where the class objects need to share information – in such cases you can specify one or more class variable and methods that are shared among the objects. When you declare a method in a class static you do not need an instance of the class. Or when you declare a variable as static than only one memory location is created for that variable among the different objects of the class.

You can use static methods to access other static members of the class, however, you cannot use the static method to access any non-static variables or methods. A class containing static members is called as a static class. There is no static identifier for the class.

Example:

```
public Class MyClass{
        public static String prop1;
        public static method1(){
                prop1 = “Hello”;
}
public static void main(String args[]){
                MyClass.prop1 = “Bye”;
MyClass.method1();
}
}
```

Java Packages

Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to.

Introduction to Java API packages:

1) java.lang:

- This package provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class which is instances of all classes at run time.
- Frequently it is necessary to represent a value of primitive type as if it were an object. The wrapper classes Boolean, Character, Integer, Long, Float, and Double serve this purpose.
- An object of type Double, for example, contains a field whose type is double, representing that value in such a way that a reference to it can be stored in a variable of reference type. These classes also provide a number of methods for converting among primitive values.
- The class Math provides commonly used mathematical functions such as sine, cosine, and square root. The classes String and StringBuffer provide commonly used operations on character strings.

2) java.util:

- This package contains the collections framework, inherited collection of classes, event model, date and time facilities, internationalization, and miscellaneous utility classes like Stack, stringTokenizer, Calendar, random-number generator, etc.

3) java.io:

- This package provides for system input and output through data streams, serialization and the file system. This package is allowing stream into two ways one is character stream and another is byte stream.

4) java.net:

- This package provides the classes for implementing networking applications. Using the socket classes, you can communicate with any server on the Internet or implement your own Internet server.
- A number of classes are provided to make it convenient to use Universal Resource Locators (URLs) to retrieve data on the Internet. Provides the classes for implementing networking applications.

5) java.applet:

- This package provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
- The applet framework involves two entities: the *applet* and the *applet context*.
- An applet is an embeddable window (see the Panel class) with a few extra methods that the applet context can use to initialize, start, and stop the applet.
- The applet context is an application that is responsible for loading and running applets. For example, the applet context could be a Web browser or an applet development environment.

6) Java.awt:

- This package contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called in AWT terminology a component. The Component class is the root of all AWT components.
- Some components fire events when a user interacts with the components. The AWTEvent class and its subclasses are used to represent the events that AWT components can fire.
- A container is a component that can contain components and other containers. A container can also have a layout manager that controls the visual placement of components in the container. The AWT package contains several layout manager classes and an interface for building your own layout manager.

1) java.lang: (Object, Math, String, String Buffer, wrapper classes)

- **Object:** Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, implement the methods of this class. This class has only one constructor : object()

Method	Description
Protected Object clone()	Creates and returns a copy of this object.
Boolean equals (Object obj)	Indicates whether some other object is "equal to" this one.
Protected void finalize()	Called by the garbage collector.
class getClass()	Returns the runtime class of an object.
int hashCode()	Returns a hash code value for the object.
void notify()	Wakes up a single thread that is waiting on this object's monitor.
Void notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String toString()	Returns a string representation of the object.
Void wait()	Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
Void wait (long time)	Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

- **Math:** The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Constants	Desctiption
static double E	The double value that is closer than any other to *e*, the base of the natural logarithms.
static double PI	The double value that is closer than any other to *pi.*

Method	Description
static double abs(double a)	Returns the absolute value of a double value.
static float abs(float a)	Returns the absolute value of a float value.
static int abs(int a)	Returns the absolute value of an int value.
static long abs(long a)	Returns the absolute value of a long value.
static double acos(double a)	Returns the arc cosine of an angle, in the range of 0.0 through *pi*.
static double asin(double a)	Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.
static double atan(double a)	Returns the arc tangent of an angle, in the range of -*pi*/2 through *pi*/2.
static double atan2(double y, double x)	Converts rectangular coordinates (x, y) to polar (r, *theta*).
static double ceil(double a)	Returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical

	integer.
static double cos(double a)	Returns the trigonometric cosine of an angle.
static double exp(double a)	Returns Euler's number *e* raised to the power of a double value.
static double floor(double a)	Returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.
static double log(double a)	Returns the natural logarithm (base *e*) of a double value.
static double max(double a, double b)	Returns the greater of two double values.
static float max(float a, float b)	Returns the greater of two float values.
static int max(int a, int b)	Returns the greater of two int values.
static long max(long a, long b)	Returns the greater of two long values.
static double min(double a, double b)	Returns the smaller of two double values.
static float min(float a, float b)	Returns the smaller of two float values.
static int min(int a, int b)	Returns the smaller of two int values.
static long min(long a, long b)	Returns the smaller of two long values.
static double pow(double a, double b)	Returns the value of the first argument raised to the power of the second argument.
static double random()	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static long round(double a)	Returns the closest long to the argument.
static int round(float a)	Returns the closest int to the argument.
static double sin(double a)	Returns the trigonometric sine of an angle.
static double sqrt(double a)	Returns the correctly rounded positive square root of a double value.
static double tan(double a)	Returns the trigonometric tangent of an angle.
static double toDegrees(double angrad)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
static double toRadians(double angdeg)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Ex.

```
Class MethDemo
{
        public static void main(String s[])
        {
                System.out.println("Value of The PI is :-"+Math.PI);
                System.out.println("Value of The E is :-"+Math.E);
                System.out.println("Absulate of 12.67 is :-"+Math.abs(12.67));
```

```
        System.out.println("The ceil of the 35.89 is :-"+Math.ceil(35.89));
        System.out.println("The floor value of 35.89 is :-"+Math.floor(35.89));
        System.out.println("3 to power of 4 is :-"+Math.pow(3,4));
    }
}
```

- **String:** The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class. Strings are constant; their values cannot be changed after they are created. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

Constructor	Description
String()	Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes)	Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, int offset, int length)	Constructs a new String by decoding the specified sub array of bytes using the platform's default charset.
String(char[] value)	Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
String(char[] value, int offset, int count)	Allocates a new String that contains characters from a subarray of the character array argument.
String(String original)	Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

Method	Description
char charAt(int index)	Returns the character at the specified index.
int compareTo(Object o)	Compares this String to another Object.
int compareTo(String anotherString)	Compares two strings lexicographically.
int compareToIgnoreCase(String str)	Compares two strings with ignoring case differences.
String concat(String str)	Concatenates the specified string to the end of this string.
boolean contentEquals(StringBuffer sb)	Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
static String copyValueOf(char[] data)	Returns a String that represents the character array specified.
static String copyValueOf(char[] data, int offset, int count)	Returns a String that represents the character array specified.
boolean endsWith(String suffix)	Tests if this string ends with the specified suffix.
boolean equals(Object anObject)	Compares this string to the specified object.
Boolean equalsIgnoreCase(String anotherString)	Compares this String to anotherString with ignoring case.

byte[] getBytes()	Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copies characters from this string into the destination character array.
int hashCode()	Returns a hash code for this string.
Int indexOf(int ch)	Returns the index of the first occurrence of the specified character.
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.
int indexOf(String str)	Returns the index of the first occurrence of the specified substring.
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.
int lastIndexOf(int ch)	Returns the index of the last occurrence of the specified character.
int lastIndexOf(int ch, int fromIndex)	Returns the index of the last occurrence of the specified character, searching backward starting at the specified index.
int lastIndexOf(String str)	Returns the index of the rightmost occurrence of the specified substring.
int lastIndexOf(String str, int fromIndex)	Returns the index of the last occurrence of the specified substring, searching backward starting at the specified index.
int length()	Returns the length of this string.
boolean matches(String regex)	Tells whether or not this string matches the given regular expression.
boolean regionMatches(int toffset, String other, int ooffset, int len)	Tests if two string regions are equal.
String replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement.
String replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement.
String[]split(String regex)	Splits this string around matches of the given regular expression.
String[]split(String regex, int limit)	Splits this string around matches of the given regular expression.
boolean startsWith(String prefix)	Tests if this string starts with the specified prefix.
boolean startsWith(String prefix, int toffset)	Tests if this string starts with the specified prefix beginning a specified index.
String substring(int beginIndex)	Returns a new string that is a substring of this string.
String substring(int beginIndex,	Returns a new string that is a substring of this string.

int endIndex)	
char[]toCharArray()	Converts this string to a new character array.
String toLowerCase()	Converts all of the characters in this String to lower case.
String toString()	This object (which is already a string!) is itself returned.
String toUpperCase()	Converts all of the characters in this String to upper case
String trim()	Returns a copy of the string, with leading and trailing whitespace omitted.
static String valueOf(boolean b)	Returns the string representation of the boolean argument.
static String valueOf(char c)	Returns the string representation of the char argument.
static String valueOf(char[] data)	Returns the string representation of the char array argument.
static String valueOf(char[] data, int offset, int count)	Returns the string representation of a specific subarray of the char array argument.
static String valueOf(double d)	Returns the string representation of the double argument.
static String valueOf(float f)	Returns the string representation of the float argument.
static String valueOf(int i)	Returns the string representation of the int argument.
static String valueOf(long l)	Returns the string representation of the long argument.
static String valueOf(Object obj)	Returns the string representation of the Object argument.

Ex.

```
Class StringDemo
{
        public staitic void main(String s[])
        {
                String s1=new String("God is great");
                System.out.println("Character of 5th position is :"+s1.charAt(5));
                System.out.println("Add into string is :"+s1.concat("add"));
                System.out.println("Is String ends with ('add')?="+s1.endsWith("add"));
                System.out.println("Is String start with ('add')?="+s1.startsWith("God"));
                System.out.println("The Length of the strig is :="+s1.length());
                System.out.println("substring of s1 is :-"+s1.subString(5));
                System.out.println("String into lower case is :-"+s1.lowerCase());
                System.out.println("String into Upper case"+s1.uppercase());
        }
}
```

- **StringBuffer:** A string buffer implements a mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls. String Buffer class is automatic allocate the 16 bytes extra for further changes into string E.g Is the string contains "this" then it occupies 20 bytes into memory.

Constructor	**Description**
StringBuffer()	Constructs a string buffer with no characters and initial capacity of 16 characters.

StringBuffer(int length)	Constructs a string buffer with no characters in it and an initial capacity specified by the length argument.
StringBuffer(String str)	Constructs a string buffer so that it represents the same sequence of characters as the string argument; in other words, the initial contents of the string buffer are a copy of the argument string.

Method	**Description**
StringBuffer append(boolean b)	Appends the string representation of the boolean argument.
StringBuffer append(char c)	Appends the string representation of the char argument.
StringBuffer append(char[] str)	Appends the string representation of the char array argument.
StringBuffer append(char[] str, int offset, int len)	Appends the string representation of a subarray of the char array argument.
StringBuffer append(double d)	Appends the string representation of the double argument.
StringBuffer append(float f)	Appends the string representation of the float argument.
StringBuffer append(int i)	Appends the string representation of the int argument.
StringBuffer append(long l)	Appends the string representation of the long argument.
StringBuffer append(Object obj)	Appends the string representation of the Object argument.
StringBuffer append(String str)	Appends the string to this string buffer.
StringBuffer append(StringBuffer sb)	Appends the specified StringBuffer to this StringBuffer.
Int capacity()	Returns the current capacity of the String buffer.
Char charAt(int index)	The specified character of the sequence currently represented by the string buffer, as indicated by the index argument.
StringBuffer delete(int start, int end)	Removes the characters in a substring of this StringBuffer.
StringBuffer deleteCharAt(int index)	Removes the character at the specified position.
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Characters are copied from this string buffer into the destination character array dst.
Int indexOf(String str)	Returns the index within this string of the first occurrence of the specified substring.
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.
StringBuffer insert(int offset, boolean b)	Inserts the string representation of the boolean argument into this string buffer.
StringBuffer insert(int offset, char c)	Inserts the string representation of the char argument into this string buffer.
StringBuffer insert(int offset, char[] str)	Inserts the string representation of the char array argument into this string buffer.
StringBuffer insert(int index, char[] str, int offset, int len)	Inserts the string representation of a subarray of the str array argument into this string buffer.

StringBuffer insert(int offset, String str)	Inserts the string into this string buffer.
Int lastIndexOf(String str)	Returns the index within this string of the rightmost occurrence of the specified substring.
Int lastIndexOf(String str, int fromIndex)	Returns the index within this string of the last occurrence of the specified substring.
Int length()	Returns the length (character count) of this string buffer.
StringBuffer replace(int start, int end, String str)	Replaces the characters in a substring of this StringBuffer with characters in the specified String.
StringBuffer reverse()	The character sequence contained in this string buffer is replaced by the reverse of the sequence.
void setCharAt(int index, char ch)	The character at the index of this string buffer is set to ch.
void setLength(int newLength)	Sets the length of this String buffer.
String substring(int start)	Returns a new String that contains a subsequence of characters currently contained in this StringBuffer. The substring begins at the specified index and to the end of the StringBuffer.
String substring(int start, int end)	Returns a new String that contains a subsequence of characters currently contained in this StringBuffer.
String toString()	Converts to a string representing the data in this string buffer.

Ex.

```
Class StringBufferDemo
{
        public static void main(String s[])
        {
                StringBuffer sb1=new StringBuffer("God is great ");
                System.out.println("Capacity is:"+sb1.capacity())
                System.out.println("Length of the string"+sb1.length());
                Sb1.setCharAt(4,'4') ;
                System.out.println("Now sb1 is"+sb1);
                System.out.println("Delete from 5 to 10 is "+sb1.delete(5,10));
                System.out.println("Delete character at 4th is:-"+sb1.deleteCharAt(4));
                System.out.println("Reverse is :-"+sb1.reverse());
        }
}
```

- **Wrapper:** Wrapper objects are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed. All wrapper classes (except Character) define a static method called valueOf(), which returns the wrapper object corresponding to the primitive value represented by the String argument. All the numeric wrapper classes have six methods, which can be used to convert a numeric wrapper to any primitive numeric type. These methods are byteValue, doubleValue, floatValue, intValue, longValue and shortValue.

  Parser methods: The six parser methods are parseInt, parseDouble, parseFloat, parseLong, parseByte and parseShort. They take a String as the argument and convert it to the corresponding primitive. They throw a NumberFormatException if the String is not properly formed.

2) java.util package classes : (Date, Random, Calendar, GregorianCalendar, Vector, enumeration

interface, Stack, Hashtable, StreamTokenizer)

- **Date:** The Date class encapsulates the current date and time. It is mainly used to manipulate dates and time. This class supports the constructors: 1) Date () 2) Date (long *millisec*)
- The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Method	Description
boolean after(Date *date*)	This method returns true if the invoking Date object contains a date that is later that the one specified by *date*. Otherwise, it returns false.
boolean before(Date *date*)	This method returns true if the invoking Date object contains a date that is earlier that the one specified by *date*. Otherwise, it returns false.
Int comperTo(Date *date*)	This method compares the value of the invoking object with that of *date*. It will return 0 if the values are equals. It will return a negative value if the invoking object is earlier than *date otherwise* return a positive value.
Int comperTo(Object *obj*)	It operates identically to compareTo(Date) if *obj* is of class Date. Otherwise, it throws a ClassCastException.
boolean equals(Object *date*)	This method returns true if the invoking Date object contains the same time and date as the one specified by *date*. Otherwise, it returns false.
long getTime()	This method returns the number of milliseconds that have elapsed since January 1, 1970.
Void setTime(long time)	This method sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January,1, 1970.

Ex.

```
Class DateDemo
{
        public static void main(String s[])
        {
                Date date=new Date()
                System.out.println(date);
                long  msec=date.getTime();
        System.out.println("Milliseconds since Jan 1,1970 GMT="+msec);
        }
}
```

- **Random:** This class is used to generate random number. This class defines two constructors Random() and Random(long seed). This first version creates a number generator that uses the current time as the starting, or seed value. The second form allows you to specify a seed value manually. If you initiaze a Random object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another Random object, extract the same random sequence.

Method	Description
Boolean nextBoolean()	This method returns the next boolean number.
Void nextBytes(byte *vals*[])	This method fills *vals* with randomly generated values.
Double nextDouble()	This method returns the next double random number.
Float nextFloat()	This method returns the next float random number.
Double nextGaussian()	This method returns the next Gaussian random number.
Int nextInt()	This method returns the next int random number.
int nextInt(int n)	This method returns the next int random number within range zero to n.
Long nextLong()	This method returns the next long random number
Void setSeed(long *newSeed*)	This method sets the seed value(that is, the starting point for the random number generator) to that specified by *newSeed*.

Ex.

```
Class RandomDemo
{
        public static void main(String s[])
        {
                Random rnd = new Random();
        int I,j;
        for(I=0;I<10;I++)
        {
                j=rnd.nextInt(100);
                System.out.println(j);
        }
        }
}
```

- **Calendar:** Calendar is an abstract base class for converting between a Date object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on. Like other locale-sensitive classes, Calendar provides a class method, getInstance, for getting a generally useful object of this type. Calendar's getInstance method returns a Calendar object whose time fields have been initialized with the current date and time:

```
Calendar rightNow = Calendar.getInstance();
```

Fields	Description
static int AM	Value of the AM_PM field indicating the period of the day from midnight to just before noon.
static int AM_PM	Field number for get and set indicating whether the HOUR is before or after noon.
static int APRIL	Value of the MONTH field indicating the fourth month of the year.
static int AUGUST	Value of the MONTH field indicating the eighth month of the year.

static int DATE	Field number for get and set indicating the day of the month.
static int DAY_OF_MONTH	Field number for get and set indicating the day of the month.
static int DAY_OF_WEEK	Field number for get and set indicating the day of the week.
static int DAY_OF_WEEK_IN_MONTH	Field number for get and set indicating the ordinal number of the day of the week within the current month.
static int DAY_OF_YEAR	Field number for get and set indicating the day number within the current year.
Static int DECEMBER	Value of the MONTH field indicating the twelfth month of the year.
static int DST_OFFSET	Field number for get and set indicating the daylight savings offset in milliseconds.
static int ERA	Field number for get and set indicating the era, e.g., AD or BC in the Julian calendar.
static int FEBRUARY	Value of the MONTH field indicating the second month of the year.
static int FIELD_COUNT	The number of distinct fields recognized by get and set.
protected int[]fields	The field values for the currently set time for this calendar.
static int FRIDAY	Value of the DAY_OF_WEEK field indicating Friday.
static int HOUR	Field number for get and set indicating the hour of the morning or afternoon.
static int HOUR_OF_DAY	Field number for get and set indicating the hour of the day.
protected boolean[] isSet	The flags which tell if a specified time field for the calendar is set.
protected Boolean isTimeSet	True if then the value of time is valid.
static int JANUARY	Value of the MONTH field indicating the first month of the year.
static int JULY	Value of the MONTH field indicating the seventh month of the year.
static int JUNE	Value of the MONTH field indicating the sixth month of the year.
static int MARCH	Value of the MONTH field indicating the third month of the year.
static int MAY	Value of the MONTH field indicating the fifth month of the year.
static int MILLISECOND	Field number for get and set indicating the millisecond within the second.
static int MINUTE	Field number for get and set indicating the minute within the hour.
static int MONDAY	Value of the DAY_OF_WEEK field indicating Monday.
static int MONTH	Field number for get and set indicating the month.
Static int NOVEMBER	Value of the MONTH field indicating the eleventh month of the year.
static int OCTOBER	Value of the MONTH field indicating the tenth month of the year.
static int PM	Value of the AM_PM field indicating the period of the day from noon to just before midnight.
Static int SATURDAY	Value of the DAY_OF_WEEK field indicating Saturday.
static int SECOND	Field number for get and set indicating the second within the minute.

static int SEPTEMBER	Value of the MONTH field indicating the ninth month of the year.
static int SUNDAY	Value of the DAY_OF_WEEK field indicating Sunday.
Static int THURSDAY	Value of the DAY_OF_WEEK field indicating Thursday.
protected long time	The currently set time for this calendar, expressed in milliseconds after January 1, 1970, 0:00:00 GMT.
static int TUESDAY	Value of the DAY_OF_WEEK field indicating Tuesday.
static int UNDECIMBER	Value of the MONTH field indicating the thirteenth month of the year.
static int WEDNESDAY	Value of the DAY_OF_WEEK field indicating Wednesday.
static int WEEK_OF_MONTH	Field number for get and set indicating the week number within the current month.
static int WEEK_OF_YEAR	Field number for get and set indicating the week number within the current year.
static int YEAR	Field number for get and set indicating the year.
static int ZONE_OFFSET	Field number for get and set indicating the raw offset from GMT in milliseconds.

Method	**Description**
abstract void add(int field, int amount)	Date Arithmetic function.
Boolean after(Object when)	Compares the time field records.
Boolean before(Object when)	Compares the time field records.
void clear()	Clears the values of all the time fields.
void clear(int field)	Clears the value in the given time field.
protected void complete()	Fills in any unset fields in the time field list.
protected abstract void computeFields()	Converts the current millisecond time value time to field values in fields[].
protected abstract void computeTime()	Converts the current field values in fields[] to the millisecond time value time.
Boolean equals(Object obj)	Compares this calendar to the specified object.
Int get(int field)	Gets the value for a given time field.
int getActualMaximum(int field)	Return the maximum value that this field could have, given the current date.
int getActualMinimum(int field)	Return the minimum value that this field could have, given the current date.
int getFirstDayOfWeek()	Gets what the first day of the week is; e.g., Sunday in US, Monday in France.
abstract int getGreatestMinimum(int field)	Gets the highest minimum value for the given field if varies.
static Calendar getInstance()	Gets a calendar using the default time zone and locale.

static Calendar getInstance(Locale aLocale)	Gets a calendar using the default time zone and specified locale.
static Calendar getInstance(TimeZone zone)	Gets a calendar using the specified time zone and default locale.
static Calendar getInstance(TimeZone zone, Locale aLocale)	Gets a calendar with the specified time zone and locale.
abstract int getLeastMaximum(int field)	Gets the lowest maximum value for the given field if varies.
abstract int getMaximum(int field)	Gets the maximum value for the given time field.
Int getMinimalDaysInFirstWeek()	Gets what the minimal days required in the first week of the year are; e.g., if the first week is defined as one that contains the first day of the first month of a year, getMinimalDaysInFirstWeek returns 1.
abstract int getMinimum(int field)	Gets the minimum value for the given time field.
Date getTime()	Gets this Calendar's current time.
long getTimeInMillis()	Gets this Calendar's current time as a long.
TimeZone getTimeZone()	Gets the time zone.
Int hashCode()	Returns a hash code for this calendar.
protected int internalGet(int field)	Gets the value for a given time field.
boolean isLenient()	Tell whether date/time interpretation is to be lenient.
boolean isSet(int field)	Determines if the given time field has a value set.
void set(int field, int value)	Sets the time field with the given value.
void set(int year, int month, int date)	Sets the values for the fields year, month, and date.
void set(int year, int month, int date, int hour, int minute)	Sets the values for the fields year, month, date, hour, and minute.
void set(int year, int month, int date, int hour, int minute, int second)	Sets the values for the fields year, month, date, hour, minute, and second.
void setFirstDayOfWeek(int value)	Sets what the first day of the week is; e.g., Sunday in US, Monday in France.
void setLenient(boolean lenient)	Specify whether or not date/time interpretation is to be lenient.
Void setMinimalDaysInFirstWeek(int value)	Sets what the minimal days required in the first week of the year are.
void setTime(Date date)	Sets this Calendar's current time with the given Date.
void setTimeInMillis(long millis)	Sets this Calendar's current time from the given long value.
void setTimeZone(TimeZone value)	Sets the time zone with the given time zone value.
String toString()	Return a string representation of this calendar.

Ex.

```
import java.util.*;
class CalendarDemo
{
        public static void main(String args[])
        {
        Calendar calendar = Calendar.getInstance();
                System.out.println(calendar.get(Calendar.YEAR));
                System.out.println(calendar.get(Calendar.HOUR));
                System.out.println(calendar.get(Calendar.HOUR_OF_DAY));
                System.out.println(calendar.get(Calendar.MINUTE));
        }
}
```

- **GregorianCalendar:** GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar used by most of the world. The standard (Gregorian) calendar has 2 eras, BC and AD. Let's see the fields provide by this class.

static int AD	Value of the ERA field indicating the common era (Anno Domini), also known as CE.
static int BC	Value of the ERA field indicating the period before the common era, also known as BCE.

Method	**Description**
void add(int field, int amount)	Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.
Void setGregorianChange(Date date)	Sets the GregorianCalendar change date.

- **Vector:** The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. Constructor of the vector class is :

  Vector()
  Vector(int initialCapacity)
  Vector(int initialCapacity, int capacityIncrement)

Field	**Description**
protected int capacityIncrement	The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
protected int elementCount	The numbers of valid components in this Vector object.
protected Object[] elementData	The array buffer into which the components of the vector are stored.

Method	**Description**

void add(int index, Object element)	Inserts the specified element at the specified position in this Vector.
Boolean add(Object o)	Appends the specified element to the end of this Vector.
void addElement(Object obj)	Adds the specified component to the end of this vector.
int capacity()	Returns the current capacity of this vector.
void clear()	Removes all of the elements from this Vector.
Boolean contains(Object elem)	Tests if the specified object is a component in this vector.
void copyInto(Object[] anArray)	Copies the components of this vector into the specified array.
Object elementAt(int index)	Returns the component at the specified index.
Enumeration elements()	Returns an enumeration of the components of this vector.
Void ensureCapacity(int minCapacity)	Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
Boolean equals(Object o)	Compares the specified Object with this Vector for equality.
Object firstElement()	Returns the first component (the item at index 0) of this vector.
Object get(int index)	Returns the element at the specified position in this Vector.
int hashCode()	Returns the hash code value for this Vector.
int indexOf(Object elem)	Searches for the first occurence of the given argument, testing for equality using the equals method.
int indexOf(Object elem, int index)	Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
void insertElementAt(Object obj, int index)	Inserts the specified object as a component in this vector at the specified index.
Boolean isEmpty()	Tests if this vector has no components.
Object lastElement()	Returns the last component of the vector.
int lastIndexOf(Object elem)	Returns the index of the last occurrence of the specified object.
int lastIndexOf(Object elem, int index)	Searches backwards for the specified object, starting from the specified index, and returns an index to it.
Object remove(int index)	Removes the element at the specified position in this Vector.
Boolean remove(Object o)	Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
Void removeAllElements()	Removes all components from this vector and sets its size to zero.
Boolean removeElement(Object obj)	Removes the first (lowest-indexed) occurrence of the argument.
void removeElementAt(int index)	Deletes the component at the specified index.

protected void removeRange(int fromIndex,int toIndex)	Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
Object set(int index, Object element)	Replaces the element at the specified position in this Vector with the specified element.
void setElementAt(Object obj, int index)	Sets the component at the specified index of this vector to be the specified object.
void setSize(int newSize)	Sets the size of this vector.
int size()	Returns the number of components in this vector.
Object[]toArray()	Returns an array containing all of the elements in this Vector in the correct order.
Object[] toArray(Object[] a)	Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
Void trimToSize()	Trims the capacity of this vector to be the vector's current size.

Ex.

```
Import java.util.*;
Class VectorDemo
{
        public static void main(String s[])
        {
                Vector vector=new Vector();
                Vector.addElement(new Integer(5));
                Vector.addElement(new Float(-14.45f));
                Vector.addElement(new String(Hello));
                Vector.addElement(new Long(12000000000));
                Vector.addElement(new Double(-23.45e-11));
                System.out.println(vector);
                String s1 = new String("String inserted");
                Vector.add(1,s1);
                System.out.println(vector);
                Vector.removeElementAt(3);
                System.out.println("After removing 3rd element ");
                System.out.println(vector);
        }
}
```

- **Enumeration interface:** An object that implements the Enumeration interface generates a series of elements, one at a time.

Method	**Description**
Boolean hasMoreElements()	Tests if this enumeration contains more elements.
Object nextElement()	Returns the next element of this enumeration if this enumeration object has

	at least one more element to provide.

Ex.

```
Import java.util.*;
Class VectorDemo
{
        public static void main(String s[])
        {
                Vector vector=new Vector();
                vector.addElement(new Integer(5));
                vector.addElement(new Float(-14.45f));
                vector.addElement(new String(Hello));
                vector.addElement(new Long(12000000000));
                vector.addElement(new Double(-23.45e-11));
               for (Enumeration e=vector.elements();e.hasMoreElements();)
                        System.out.println(e.nextElement());
        }
}
```

- **Stack:** The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. When a stack is first created, it contains no items. It has one constructor : Stack()

Method	**Description**
Boolean empty()	Tests if this stack is empty.
Object peek()	Looks at the object at the top of this stack without removing it from the stack.
Object pop()	Removes the object at the top of this stack and returns that object.
Object push(Object item)	Pushes an item onto the top of this stack.
int search(Object o)	Returns the position where an object is on this stack.

Ex.

```
import java.util.*;
Class StackDemo {
        public static void main(String s[]) {
                Stack S1= new Stack();
                S1.push(new Integer(10));
                S1.push(new Integer(20));
                S1.push(new Integer(30));
                S1.push(new Integer(40));
                System.out.println("Is Stack Empty :"+S1.empty());
                System.out.println("Top of The Stack is :"+S1.peek());
                System.out.println("Remove from the stack :"+S1.pop());
                System.out.println("Data of stack :"+S1);
        }
}
```

- **Hashtable :** This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value. This class provides following constructure:

  Hashtable()
  Hashtable(int initialCapacity)
  Hashtable(int initialCapacity, float loadFactor)
  Hashtable(Map t)

Method	**Description**
void clear()	Clears this hashtable so that it contains no keys.
Object clone()	Creates a shallow copy of this hashtable.
Boolean contains(Object value)	Tests if some key maps into the specified value in this hashtable.
Boolean containsKey(Object key)	Tests if the specified object is a key in this hashtable.
Boolean containsValue(Object value)	Returns true if this Hashtable maps one or more keys to this value.
Enumeration elements()	Returns an enumeration of the values in this hashtable.
Boolean equals(Object o)	Compares the specified Object with this Map for equality.
Object get(Object key)	Returns the value to which the specified key is mapped.
Boolean isEmpty()	Tests if this hashtable maps no keys to values.
Enumeration keys()	Returns an enumeration of the keys in this hashtable.
Object put(Object key, Object value)	Maps the specified key to the specified value in this hashtable.
Void putAll(Map t)	Copies all of the mappings from the specified Map to this Hashtable. These mappings will replace any mappings that this Hashtable had for any of the keys currently in the specified Map.
protected void rehash()	Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently.
Object remove(Object key)	Removes the key (and its corresponding value) from this hashtable.
int size()	Returns the number of keys in this hashtable.
String toString()	Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space).

Ex.

```
import java.util.*;
Class HashClass
{
        public static void main(String s[])
        {
                Hashtable h1=new Hashtable();
                System.out.println(" Is Table empty ?:"+h1.isEmpty());
                h1.put("the color of Apple is","red");
```

```
                h1.put("the color of banana is","yellow");
                System.ot.println("Data of HashTable:"+h1);
                System.ot.println("Value of red is:"+h1.get("red"));

                System.out.println(" Is Table empty ?:"+h1.isEmpty());
                System.out.println("Remove from table is :"+h1.remove("The color of apple is"));
                System.out.println("Now Data of Hash table is :",h1");
        }
    }
```

- **StringTokenizer:** The StringTokenizer class allows an application to break a string into tokens. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments. A StringTokenizer object internally maintains a current position within the string to be tokenized. A token is returned by taking a substring of the string that was used to create the StringTokenizer object. There are three constructors:

  StringTokenizer(String str)
  StringTokenizer(String str,String delim)
  StringTokenizer(String str,String delim,boolean returnDelims)

- The first constructor constructs a string tokenizer for the specified string. Second constructor constructs a string tokenizer for the specified string with specific delimiter. In the third constructor, if returnDelims as true than the delimiters are also returned Otherwise the delimiters are not returned.

Method	**Description**
int countTokens()	Calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.
boolean hasMoreElements()	Returns the same value as the hasMoreTokens method.
boolean hasMoreTokens()	Tests if there are more tokens available from this tokenizer's string.
Object nextElement()	Returns the same value as the nextToken method, except that its declared return value is Object rather than String.
String nextToken()	Returns the next token from this string tokenizer.
String nextToken(String delim)	Returns the next token in this string tokenizer's string

Ex.

```
import java.util.*;
class StringTokenDemo
{
        public static void main(String s[])
        {
                StringTokenizer st = new StringTokenizer("this is a test");
                while (st.hasMoreTokens())
                {
                        System.out.println(st.nextToken());
                }
        }
}
```

Output:

this
is
a
test

Creating Own Packages:

- To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package which has no name.
- This is the general form of the package statement. Package statement is first statement of the program.

```
package pkg;
class First
{
        : : :
}
```

- Here class First now the part of the pkg package. This package is must store into the pkg directory. Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.
- It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create hierarchy of the packages.
- If you want to use developed packages then you have to define one environment variable named CLASSPATH. This variable is set up to package directory.
- If you created one package pkg into c:\Mypackage directory then you have to set CLASSPATH variable to "c:\Mypackage". Now you can access all classes of the pkg package from any of the path.
- We can also import the created packages using same as inbuilt packages like below.

```
import pkg.*;
```

- Using import keyword any package is imported into any program. This is illustrated below.

Ex.

```
package pkg;
public class Balance {
    String name;
    double amt;
    public Balance(String s, double d) {
            Name = s;
            amt = d;
    }
    public void disp()
    {
            System.out.println("Display from Balance of pkg package");
            System.out.println(name + " has " + amt + " Balance " );
    }   }
```

- This program is write into Balance.java file because public class is stored into file which is same as class name and only one class is public in one java program.
- When we compile this program then it will generate one package named pkg and one class named Balance into that package. So, Balance is part of the pkg package.

UNIT-3

Exception Handling, Threading and Streams(Input and Output)

EXCEPTION HANDLING

- A java exception is an object that describes an exceptional(error) condition that has occurred in piece of code.
- They could be file not found exception, unable to get connection exception and so on.
- Java exception handling is managed via 5 keywords :
    - **try**

      The java code that you think may produce an exception is placed within a try block for a suitable catch block to handle the error. If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error. Again if the matching catch handler is not found execution proceeds with the finally block and the default exception handler throws an exception.

    - **catch**

Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed

    - **throw**

      To manually throw an exception use the keyword **throw.** It is used to raise an exception within the program, the statement would be throw new exception.

      Syntax :- throw ThrowableInstance;

    - **throws**

      throws clause is used to indicate the exceptions that are not handled by the method. It must specify this behavior so the callers of the method can guard against the exceptions. It is specified in the methods signature.

      Syntax :-
      ```
      type methodname(arglist) throws exception-list
      {
          //body of method
      }
      ```
      If there are multiple exceptions, they are separated by a comma.

    - **finally**

      A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Generally finally block is used for freeing resources, cleaning up, closing connections etc. If the finally clock executes a control transfer statement such as a return or a break statement, then this control statement determines how the execution will proceed regardless of any return or control statement present in the try or catch.

- Syntax:-

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

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

Exceptions Types in Java

The super class of all class is Throwable class. Error and Exception are two sublcasses of Throwable class. The Exception class has subclasses that our program can catch. This class is also used to create our own exception. This can be done by extending the Exception class. Error class defines some internal errors that are not supposed to catch by user. An example of error is Stack Overflow.

There are two sublcasses of Exception class: (1) RuntimeException and (2) IOException. Examples of RuntimeException includes array index out of bounds or division by zero etc. An example of IOException are trying to read past to the end of a file etc.

Here, is a simple example that shows what happens when exception is not handled. This example tries to access an element of array which is out of limit.

```
//this program has a runtime exception….
class exceptionex {
public static void main(String args[])
{
        int arr[]={23,44,55};
        System.out.println("arr[25]= "+arr[25]);//array index out of bounds
}
}
```

In this program, we have tried to access the array element which is out of bounds. The size of array is 3 elements. So we can not access beyond its limits. We have tried to access 26^{th} element of the array. So there is a runtime exception named ArrayIndexOutOfBoundsException. The output shows it. It shows in which method (main(),here) , at which line number and which exception has occurred. This exception class is in java.lang package. This program is compiled successfully but at the runtime an exception occurs and the program is terminated undesirably.

- Exception
  - This class is used for exceptional conditions that user programs should catch.
  - The class Exception represents exceptions that a program faces due to abnormal or special conditions during execution. Exceptions can be of 2 types: Checked (Compile time Exceptions)/ Unchecked (Run time Exceptions).
  - Checked exception :-
    - Exceptions that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called checked exception.

ClassNotFoundExcption	Class not found
CloneNotSupportedExcption	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is defined.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist.

- **Unchecked exception** :-

ArrayIndexOutOfBoundsException	Array index out of bounds
ArrayStoreException	Assignement to an array element of and incompatible type
ClassCastException	Invalid cast
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as writing on an unlocked thread
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference
NumberFormatException	Invalid conversion of a string to a numeric format
SecurityException	Attempt to violate security
StringIndexOutOfBounds	Attempt to index outside the bounds of a string
UnsupportedOperationException	An unsupported operation was encountered.

Multithreading, the Thread class, and the Runnable interface

Java Thread Model

It allows your computer to run two or more programs concurrently. Java supports multiple threads to run concurrently which is known as multithreading. This is far more advantages than single threaded systems. In a single threaded system, a thread runs for indefinite time. While a thread is running, it may fire some event and wait for a signal. So between two events. It stops for some time. This is called a CPU cycle. Thus the time for these CPU cycles are wasted. But if we have multiple threads, then if a thread holds for some interval of time, other threads are running. So the wasted CPU cycle time can be eliminated.

Multithreading or JAVA Thread model:-

- Is a fundamental feature of the Java language specification. The creators of Java understood the importance of multithreading and provided easy to use features for application developers. Two of these features, the Thread class and the Runnable interface, will be covered shortly.
- It means a single program can perform two or more task simultaneously.
- A multithreaded program contains two or more parts that can run simultaneously. Each part of such program is called a **thread,** and each thread defines a separate path of execution.
- The java runtime system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- Single threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with a signal that a network file is ready to read, then event loop dispatches control to the appropriate event handler. This wastes CPU time.
- Is not the same as multiprocessing. The latter involves the use of two or more processors (CPUs). Multithreading involves the sharing of a single processor and the interleaving of CPU time as shown by the following diagram:

	CPU time
Thread 1	Xxxxxxxxxx xxxxx
Thread 2	xxxxxxxxxxxxxxx xxxxxxxx xxxxxx
Thread 3	xxxxxxx xxxxxxxxxx

- Because there is only one processor, only one instruction can be executed at a time. In a Java program, the decision as to which thread is currently executing is determined by the JVM.
- Is used to execute even the simplest Java program. Even if a program doesn't create its own threads, the Java Virtual Machine creates multiple threads to support the program. One thread performs the processing of the main() method. Other threads manage and monitor system resources. The garbage collector, for example, is always running as a low-priority thread.

The life cycle of a thread :-

1] Newborn State :-

➔ When we create a thread object, the thread is born and is said to be in newborn state.

➔ At this state, we can do only one of the following things with it.

- Schedule it for running using **start()** method.
- Kill it using **stop()** method.

2] Runnable State :-

➔ The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution.

➔ If all threads have equal priority, then they are given time slots for execution in round robin fashion i.e. first come first serve manner.

➔ The process of assigning time to threads is known as **time-slicing.**

➔ However, if we want to relinquish control to another thread of equal priority before it turn comes, we can do so by using the **yield()** method.

3] Running State :-

➔ Running means that the processor has given its time to the thread for its execution.

➔ The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

➔ A running thread may relinquish its control in one of the following situations :

- It has been suspended using **suspend()** method. A suspended thread can be revived by using the **resume()** method.
  It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep(time)** where time is in milliseconds.
- It has been told to wait until some event occurs. This is done using wait() method. The thread can be scheduled to run again using the notify() method.

4] Blocked State :-

➔ A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state.

➔ This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.

➔ A blocked thread is considered " not runnable" but not dead and therefore fully qualified to run again.

5] Dead State :-

➔ Every thread has a life cycle.

➔ A running thread ends its life when it has completed executing its run() method. It is a natural death.

➔ However, we can kill it by sending the stop message to it any state thus causing a premature death to it.

➔ A thread can be killed as soon it is born, or while it is running, or even when it is in " not runnable" condition.

The Thread class

- Is part of the java.lang package so no import statement is required
- Is an extension of the Object class
- Encapsulates the processing of a thread
- Can be extended by a class that must process as a thread. All that is required is to override the inherited run() method to define the thread's processing. For example,

```
public class GetMad extends Thread {
public void run() {
for (int i = 1; i <= 10; i++) {
System.out.println(i);
}
System.out.println("DONE");
}
}
```

defines a class to count from 1 to 10 as a thread. For an application to create and launch the thread, all that must be coded are the statements:

```
GetMad t = new GetMad();
t.start();
```

The inherited start() method of the Thread class automatically calls the run() method to trigger the thread's processing. A program should never call the run() method directly. Once the thread has been launched, the application can continue its own processing thread with the JVM scheduling the use of the CPU by the two threads. Here is a complete example using a static inner class:

```
public class App {

public static class GetMad extends Thread {
public void run() {
for (int i = 1; i <= 10; i++) {
System.out.println(i);
}
}
}
public static void main(String[] args) {
GetMad t = new GetMad();
t.start();
for (int i = 1; i <= 100; i++) {
System.out.println("Working");
}
}
}
```

When this is run, note how the output of the two threads is interleaved.

- Has several constructors. The most frequently used is demonstrated by

```
Thread aThread = new Thread(threadObject);
```

where threadObject is the reference of an object that either extends the Thread class or implements the Runnable interface.
Using this technique, an alternative way to have constructed the thread of the previous sample would be

```
Thread t = new Thread(new GetMad());
```

- Has a number of methods, some of which are static. The most frequently used methods are the following:

Method	Usage
destroy()	A method that may be overridden to define what is to be done when the thread is to be destroyed
getName()	Returns a String representing the thread's name
getPriority()	Returns an int from 1 to 10 that indicates the thread's priority. The highest priority is 10.
interrupt()	Interrupts the thread
isAlive()	Returns a boolean indicating if the thread is currently alive
isInterrupted()	Returns a boolean indicating if the thread is currently interrupted
run()	A method that may be overridden to define thread processing. It should never be called directly.
setName()	Sets the name of the thread to the specified String
setPriority()	Sets the priority of the thread to the specified int value (from 1 to 10)
sleep()	Causes the currently executing thread to sleep for a specified number of milliseconds. This is a static (class)

	method.
start()	Causes this thread to begin execution by automatically calling its run() method
yield()	Causes the currently executing thread to give up the CPU so that other threads may execute. This is a static (class) method.
join()	Wait for thread to finish.

Note that the stop(), suspend(), and resume() methods of the Thread class are deprecated and should be avoided.

The Runnable interface

➔ Create class that implement **Runnable** interface.

➔ Implement **run()** method.

```
Public void run()
{
        -------
        -------
}
```

➔ Instantiate an object of type Thread from within that class.

Constructor :-

Thread(Runnable threadObj, String threadName)

➔ After a new thread is created, it will not start running until you call its **start()** method.

Thread Priorities :-

➔ Thread priorities are used by thread scheduler to decide when each thread should be allowed to run.

➔ Higher priority threads get more CPU time then lower priority threads.

➔ To set a thread's priority, use setPriority() method.

Syntax :-

final void setPriority(int level)

here, level must be within range of MIN_PRIORITY and MAX_PRIORITY.

MIN_PRIORITY = 1

MAX_PRIORITY=10

NORM_PRIORITY=5(default priority)

You can obtain the current priority setting by calling the getPriority() method of thread.

User Define Exception

We can also generate the exception. User define exception is generated by the Exception class and the method of the Throwable class. When we want to generate the any type of the exception which is user define or according to the application then we have to make one sub class of the Exception class.

That class is simply treated as the exception and which is user define exception. Now the object of the class is treated as object of the exception and we can throw that object as like exception in java.

Example of the user defines exception.

```
import java.lang.*;
class NoOutOfRange extends Exception
```

```
{
String msg;
NoOutOfRange(String s)
{
msg = s;
}
public String toString()
{
return("No. Out Of Range : " + msg );
}
}
class TestNoOutofRange
{
static void a(int x)
{
System.out.println("I am in Function: " + x);
if(x < 0)
throw new NoOutOfRange(" Less Than Zero ");
if(x > 100)
throw new NoOutOfRange(" More Than Hundread ");
System.out.println("Number is : "+x);
}
public static void main(String args[])
{
try
{
a(20);
a(5);
a(12);
}
catch(ArithmeticException e)
{ System.out.println(" Arithmetic Exception " + e); }
catch(NoOutOfRange e)
{ System.out.println(" User Define Exception " + e); }
catch(Exception e)
{ System.out.println(" Exception " + e); }
}
}
```

Synchronization

When more than one threads attempt to use a shared resource, a phenomenon occurs, which is known as race condition. In the programming term, " A shared resource can be a method or an object. "This race condition must be avoided because when two or more threads access a shoared resource such as a file, one may read the file while the other is writing in the file. This may lead to unexpected results. Therefore the synchronization is necessary. The synchronization ensures that only thread will access a shared resource at a time. When a thread enters a shared resource it will lock that resource and after the completion of its execution the resource will be unlocked.

The synchronization is done by the synchronized keyword. To see the need of synchronization let us see the following example which does not use synchronization but it should.

```
//this example has a non-synchronzied method…..
class printstr
{
Static void print(String s1, String s2) { //nonsynchronized method….
Sytem.out.print(s1+ " : ");
try {
        Thread.sleep(300);
}
catch(InterruptedException e){
Sytem.out.println("thread interrupted.");
}
System.out.println(s2);
}
}
class strthread implements Runnable{
Thread t;
String s1, s2;
Strthread(String s1, String s2)
{
        this.s1=s1;
        this.s2=s2;
        t=new Thread(this, "Mythread");
        t.start();
}
Public void run()
{
        PrintStr.print(s1,s2);
}
}
class nonsynchronized{
public static void main(String arg[]){
        System.out.println("Non synchronized example.");
        new Strthread("1","one");
        new Strthread("2","two");
        new Strthread("3","three");
}
}
```

Output:

```
D:\jitu>javac nonsynchronized.java
D:\jitu>java nonsynchronized
Non synchronized example.
1: 2 : 3 : one
Two
Three
D:\jitu>_
```

You can see from the output that the output is not as desired. It can be achieved by the use of synchronization. To turn this example into synchronized just change the print() method as shown below:

```
//a synchronized example….
```

```
class printstr
{
synchronized static void print(String s1, String s2) {
Sytem.out.print(s1+ " : ");
try {
        Thread.sleep(300);
}
catch(InterruptedException e){
Sytem.out.println("thread interrupted.");
}
System.out.println(s2);
}
}
class strthread implements Runnable{
Thread t;
String s1, s2;
Strthread(String s1, String s2)
{
        this.s1=s1;
        this.s2=s2;
        t=new Thread(this, "Mythread");
        t.start();
}
Public void run()
{
        PrintStr.print(s1,s2);
}
}
class synchronized{
public static void main(String arg[]){
        System.out.println("A synchronized example.");
        new Strthread("1","one");
        new Strthread("2","two");
        new Strthread("3","three");
}
}
```

Output:

```
D:\jitu>javac synchronized.java
D:\jitu>java synchronized
A synchronized example.
1: one
2: Two
3: Three
D:\jitu>_
```

Deamon Thread

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically. There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

Simple example of Daemon thread in java

File: MyThread.java

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();
    t1.setDaemon(true);//now t1 is daemon thread
  t1.start();//starting threads
 t2.start();
 t3.start();
 }
}
```

Output:

```
daemon thread work
user thread work
user thread work
```

Streams (Input and Output

The File class

- It is used to obtain and manipulate the information associated with a disk file, such as permission, time , date etc.
- Constructors :-

  File(String directoryPath)
  File(String directoryPath, String filename)
  File(File dirObj, String filename)
  File(URL urlObj**)**

In the first constructor, the path specifies the path of the file or directory. In the second constructor, the directoryPath is the path to parent directory of the file and the filename specifies the name of the file. In the third constructor, you can specify a File object of the directory and the name of the file.

- **Methods :-**

void deleteOnExit()	boolean isDirectory()
boolean isHidden()	boolean isFile()
String getName()	boolean isAbsolute()
String getParent()	int length()
String lastModified()	boolean renameTo(File newfilename)
boolean exists()	boolean delete()
boolean canRead()	boolean setLastModified(long millsec)
boolean canWrite()	boolean setReadOnly()

The file class has following two constants:

(1) Separtorchar(\) : It is the separator character that separates the directory names.

(2) pathSeparaptorChar(;) : it is the separator character that separates the paths.

Streams

A stream is a path or a medium along which the data flows. So the data passes through streams from source to destination. The source is called the input and the destination is called the output of program. The examples of input streams can be keyboard, a mouse, a file, a memory buffer etc. And the output can be a moniter screen, a printer, a file etc.

There are two main categories of streams:

(1) character Streams:

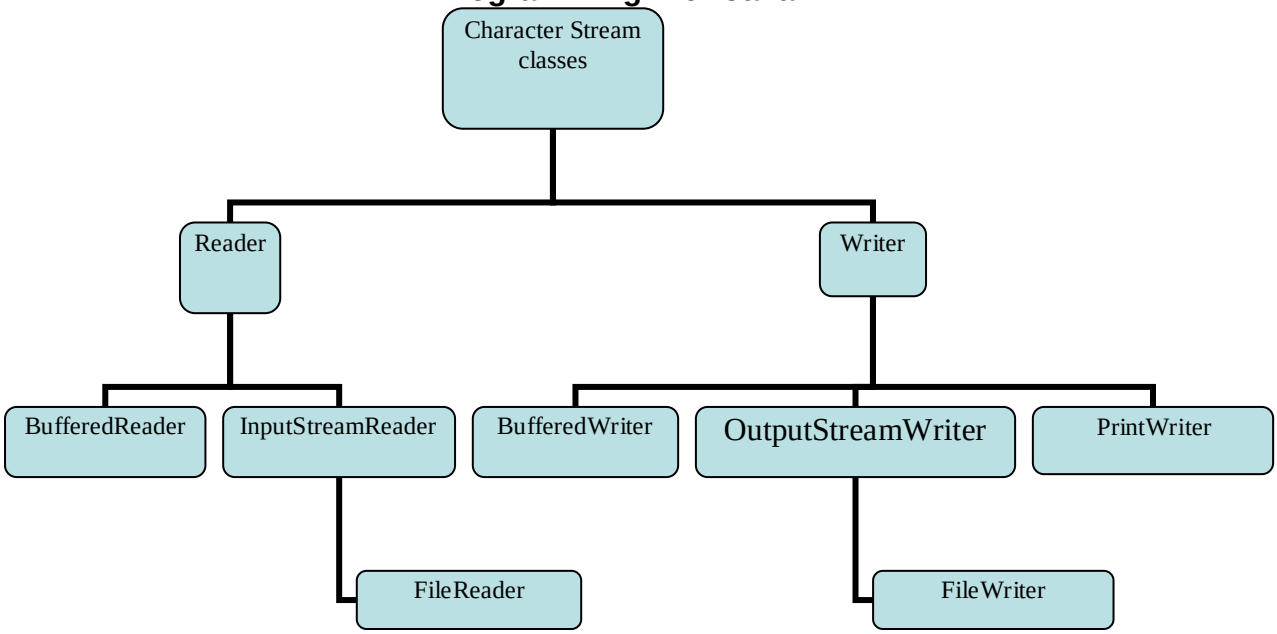
The character stream classes manipulate data as characters. A character in java is of 16-bits. Thus the character stream classes can work with 16-bit Unicode characters. The main two classes of character stream are Reader and writer.

(2) Byte streams:

The byte stream classes manipulate data byte by byte. A byte in java is of 8-bits. Thus a byte stream object reads or writes data value in binary form. The main two classes of byte stream are InputStream and OutputtSream

1. Character Streams Classes:

The character stream classes are Reader and Writer. The classes of character streams are classified as follows:



Reader :

- It is an abstract class that defines Java's model of streaming character input.
- All of the methods throws IOException.
- Methods :-

```
abstract void close()
void mark(int numChars)
boolean markSupported()
int read()
boolean ready()
void reset()
long skip(long numChars)
```

Writer class

- It is an abstract class that defines streaming character output.
- All of the methods return void and throw IOException.
- Methods :-

```
abstract void close()
abstract void flush()
void write(int ch)
void write(String str)
void write(String str, int offset, int numChars)
```

FileReader & FileWriter class

- **FileReader :-**

– The FileReader class is used to read characters from a file. It is a subclass of InputStreamReader class. Its

Constructors are:-

```
FileReader(String filePath)
```

Here, the path is the path of the input file

```
FileReader(File fileObj)
```

Here, the obj is the object of File class.

These constructors open the specified file. So if the file is not found FileNotFoundException is thrown.

- **FileWriter :-**

– It creates a Writer that you can use to write to a file.
– Constructors :-

FileWriter(String filePath)

The path specifies the path of the file to be written.

FileWriter(String filePath, boolean append)

If the second parameter append is true, the file will be opened in opened in append mode, i.e. the output will be appended to the file if the file contains some data.

FileWriter(File fileObj)

The object is an object of File class. It specifies the file to be written.

These constructors can throw an IOException.

BufferedReader & BufferedWriter classes

- **BufferedReader :-**

– The BufferedReader class is also used to take input from a stream but it increases the performance by using buffer to take input. The input is first stored in the buffer and then the data is transferred from the buffer. Its Constructors are:-

BufferedReader(Reader obj)

Here, the obj is an object of Reader's subclass. The default buffer size will be used.

BufferedReader(Reader obj, int bufferSize)

Here, you can specify the size of buffer as the second parameter.

The BufferdReader class adds a method readLine() which reads a line of characters from the keyboard. The reading of data isterminated when a new line is encountered.

Its syntax is: String readLine() throws IOException

- **BufferedWriter :-**

– It is a Writer that adds a flush() method that can be used to ensure that data buffers are physically written to the actual output stream.
– Constructors :-

BufferedWriter(Writer outputstream)
BufferedWriter(Writer outputstream, int bufSize)

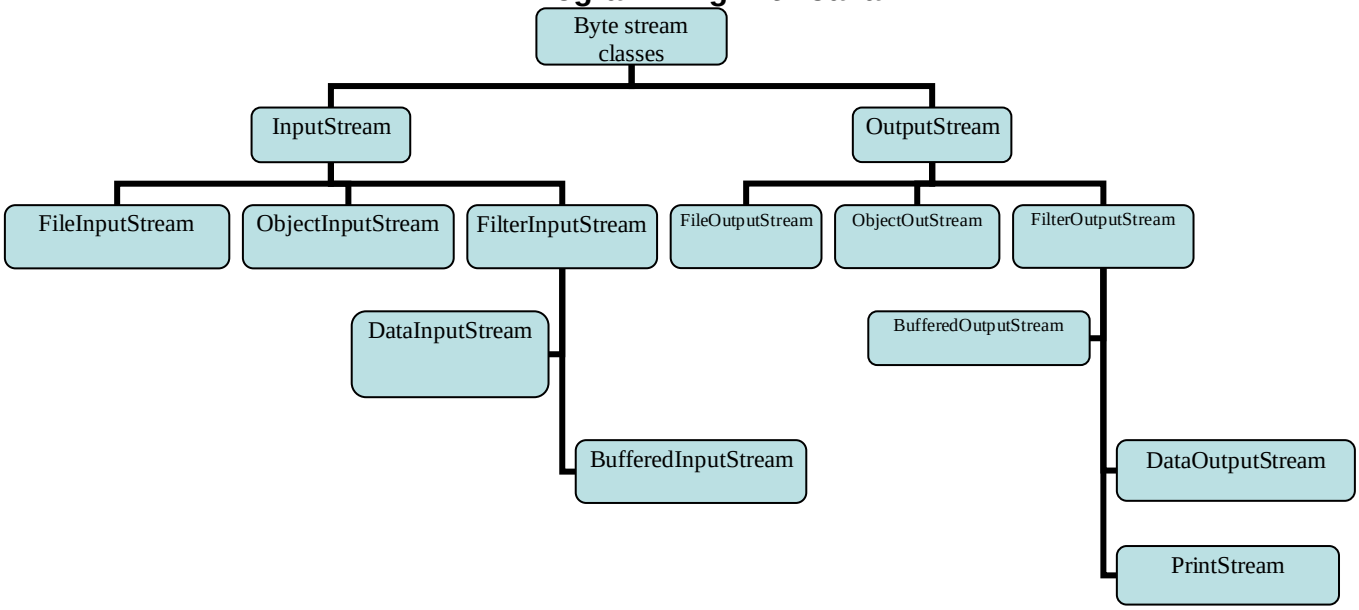
PrintWriter class

- **I**t is essentially a character-oriented version of PrintStream.
- It provides formatted output methods print() and println()
- Constructors :-

PrintWriter(OutputStream os)
PrintWriter(OutputStream os, boolean flushOnNewLine)
PrintWriter(Writer os)
PrintWriter(Writer os, boolean flushOnNewline)

The Byte Streams Classes

The byte stream classes are InputStream and OutStream. The classes of byte streams are classified as follows:



- Byte stream is used for handling byte-oriented I/O.
- It can be used with any type of object, including binary data.
- It has two super classes :

– InputStream
– OutputStream

InputStream class

- It is an abstract class that defines java's model of streaming byte input.
- All of the methods of this class will throw and IOException
- Methods :-

int available()
void close()
void mark(int numBytes)
boolean markSupported()
int read()
int read(byte buffer[])
int read(byte buffer[], int offset , int numBytes)
void reset()
long skip(long numBytes)

OuputStream class

- It is an abstract class that defines streaming byte output.
- All of the methods return void and throw IOException.
- Methods :-

void close()
void flush()
void write(int b)
void write(byte buffer[])
void write(byte buffer[] , int offset, int numBytes)

FileInputStream class

- This class creates an InputStream that you can use to read bytes from a file.
- Constructors :-

FileInputStream(String filePath)
FileInputStream(File fileObj)

FileOutputStream class

- This class creates an OutputStream that you can use to write bytes to a file.
- Constructors :-

FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)

Filtered byte streams

- Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality.
- Constructors :-

– FilterOutputStream(OutputStream os)
– FilterInputStream(InputStream is)

Buffered Byte streams

- **BufferedInputStream :-**

– It allows you to "wrap" any InputStream into a buffered stream and achieve the performance improvement.
– Constructors :-

BufferedInputStream(InputStream is)
BufferedInputStream(InputStream is,int bufsize)

- **BufferedOutputStream :-**

– It is similar to OutputStream but it added flush() method that is used to ensure that data buffers are physically written to the actual output device.
– Constructors :-

BufferedOutputStream(OutputStream os)
BufferedOutputStream(OutputStream os, int bufSize)

PrintStream class

- The PrintStream class is a subclass of FilterOutputStream class. It is used to print data as we do using print() and println() methods of Syste.out. This class also supports these methods. In these methods Java calls toString() method to convert data type to a simple data type. Its Constructors are:-

PrintStream(OutputStream obj)

The obj is an object of any OutputStream class.

PrintStream(OutputStream obj, boolean flushOnNewLine)

If the flushOnNewLine is true, the output stream is automatically gets flushed when a new line character (\n) is encountered.

Other java.io classes

Random Access File:

- It encapsulates a random access file.
- It is not derived from InputStream or OutputStrem. Instead, it implements the interfaces DataInput and DataOutput.
- It also supports positioning requests – that is, you can position the file pointer within the file.
- Constructors :-

RandomAccessFile(String filename,String access) throws FileNotFoundException
RandomAccessFile(File fileObj,String access) throws FileNotFoundException

- Methods :-
  - void seek(long newPos) throws IOException
  - void setLength(long len) throws IOException

StreamTokenizer class

streamTokenizer

Breaking a string or stream into meaningful independent words is known as tokenization. Tokenization is a common practice to tool developers. java.util package includes StringTokenizer which tokenizes a string into independent words. For StringTokenizer, the source is a string. There comes a similar tokenizer, StreamTokenize with java.io package for which source is a stream. Here, the StreamTokenizer tokenizes a whole stream into independent words.

About StreamTokenizer

For StreamTokenizer, the source is a character stream, Reader. StreamTokenizer tokenizes the stream into distinct words and allows to read one by one. It is not a general tokenizer and includes the capability of knowing the type of the token like the token is a word or number. For general tokenization, there comes java.util.Scanner where each word or line can read with next(), nextLine() and nextUTF() methods. StreamTokenizer includes a method nextToken() that can be used in a for loop to print all the tokens.

The StreamTokenizer comes with the following constant variables (instance variables are also known as fields) used to decide the type of the token.

1. int ttype: When the nextToken() returns a token, this field can be used to decide the type of the token.
2. static final int TT_EOF: This field is used to know the end of file is reached.
3. static final int TT_EOL: This field is used to know the end of line is reached.
4. static final int TT_NUMBER: This field is used to decide the token returned by the nextToken() method is a number or not.
5. static final int TT_WORD: This field is used to decide the token returned by the nextToken() method is a word or not.
6. String sval: If the token is a word, this filed contains the word that can be used in programming.
7. double nval: If the token is a word, this filed contains the number that can be used in programming

UNIT-4

Applets and Layout Managers

Applet:

- ➔ An applet is a java program that runs with the help of the web browser.
- ➔ All applets are sub-classes of the class "Applet".
- ➔ The class Applet belongs to the package "java.applet".
- ➔ To create an applet , the following two packages must be imported.
  - o java.applet
  - o java.awt

  here, awt stands for Abstract Window Toolkit)
- ➔ Applet can perform arithmetic operations, display graphics, play sounds , accept user input , create animation , and play interactive games.
- ➔ Two types of applet :
  - o Local Applet

    An applet developed locally and stored in a local system is known as a local applet.

    When a web page is trying to find a local applet , it does not need to use a internet and therefore the local system does not require the internet connection.
  - o Remote Applet

    A remote applet is that which is developed by someone else and stored on a remote computer connected to the Internet. In order to locate and load a remote applet , we must know the applet's address on the web. This address is known as URL.
- ➔ Applet can be run with :
  - o Java appletviewer
  - o Java-enabled web browser

A simple Applet program

To make an applet program you must extend Applet class. After extending Applet class you have to implements the paint() method which is in Graphics class. The Graphics class is in java.awt package(Abstract window Toolkit). It contains so many classes which are discussed later. The following creates a simple applet

```
//A simple applet program…..
import java.applet.*;
import java.awt.*;
/*
<applet code="myapplet.class" width=300 height=100>
</applet>
*/
public class myapplet extends Applet{
public void paint(Graphics g){
g.drawString("welcome to my applet,",10,25);
}
}
```

Output:

```
D:\jitu>javac myapplet.java
D:\jitu>appletviewer myapplet.java
```

<APPLET> tag :-

<APPLET
CODEBASE = *codebaseURL*
ARCHIVE = *archiveList*
CODE = *appletFile* ...or... **OBJECT = *serializedApplet***
ALT = *alternateText*
NAME = *appletInstanceName*
WIDTH = *pixels* HEIGHT = *pixels*
ALIGN = *alignment*
VSPACE = *pixels* HSPACE = *pixels*
>
<PARAM NAME = *appletAttribute1* VALUE = *value*>
<PARAM NAME = *appletAttribute2* VALUE = *value*>
. . .
alternateHTML
</APPLET>

CODE, CODEBASE, and so on are attributes of the applet tag; they give the browser information about the applet. The only mandatory attributes are CODE, WIDTH, and HEIGHT. Each attribute is described below.

CODEBASE = *codebaseURL*
This OPTIONAL attribute specifies the base URL of the applet--the directory that contains the applet's code. If this attribute is not specified, then the document's URL is used.

ARCHIVE = *archiveList*
This OPTIONAL attribute describes one or more archives containing classes and other resources that will be "preloaded". The classes are loaded using an instance of an AppletClassLoader with the given CODEBASE.
The archives in *archiveList* are separated by ",". NB: in JDK1.1, multiple APPLET tags with the same CODEBASE share the same instance of a ClassLoader. This is used by some client code to implement inter-applet communication. Future JDKs *may* provide other mechanisms for inter-applet communication. For security reasons, the applet's class loader can read only from the same codebase from which the applet was started. This means that archives in *archiveList* must be in the same directory as, or in a subdirectory of, the codebase. Entries in *archiveList* of the form ../a/b.jar will not work unless explicitly allowed for in the security policy file (except in the case of an http codebase, where archives in *archiveList* must be from the same host as the codebase, but can have ".."'s in their paths.)

CODE = *appletFile*
This REQUIRED attribute gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute. One of CODE or OBJECT must be present. The value *appletFile* can be of the form *classname*.class or of the form *packagename.classname*.class.

OBJECT = *serializedApplet*
This attribute gives the name of the file that contains a serialized representation of an Applet. The Applet will be deserialized. The init() method will *not* be invoked; but its start() method will. Attributes valid when the original object was serialized are *not* restored. Any attributes passed to this APPLET instance will be available to the Applet; we advocate very strong restraint in using this feature. An applet should be stopped before it is serialized. One of CODE or OBJECT must be present.

ALT = *alternateText*

This OPTIONAL attribute specifies any text that should be displayed if the browser understands the APPLET tag but can't run Java applets.

NAME = *appletInstanceName*

This OPTIONAL attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.

WIDTH = *pixels* HEIGHT = *pixels*

These REQUIRED attributes give the initial width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up.

ALIGN = *alignment*

This OPTIONAL attribute specifies the alignment of the applet. The possible values of this attribute are the same as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.

VSPACE = *pixels* HSPACE = *pixels*

These OPTIONAL attributes specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE). They're treated the same way as the IMG tag's VSPACE and HSPACE attributes.

<PARAM NAME = *appletAttribute1* VALUE = value>
<PARAM NAME = *appletAttribute2* VALUE = value> . . .

This tag is the only way to specify an applet-specific attribute. Applets access their attributes with the getParameter() method.

Applet life cycle :-

Each applet has a life cycle. So every time an applet is initialized it goes through four states and finally after its execution the applet is destroyed. At these states the applet calls a set of methods in an order. These methods are init(), stop(), and destroy(). These methods are defined by the Applet class and one can override these methods an needed. In addition to these four methods, the pain() method is another method which is in the component class of java.awt package. These methods are described below:

(1) The init () method:

The init() method is the first method called by an applet. In this method the initialization process is done. The variables, objects, background colors, fonts etc. are initialization should be done only once, its syntax is: public void init(){
//initialization process…
}

(2) The start() method:

After the init() method, start() method is called. This method starts the applet. it is also called to restart the applet if its stopped. This method can be called more than onece since it needs to restart each time it is stopped. In a browser if you leave the applet page, the applet is stopped and when you again come back to that page its is restarted. Or when the applet window is minimized, it is stopped and as it is restored, the applet is restarted. The syntax of start() method is:
public void start(){
//the code to start applet
}

(3) The paint() method:

The paint() method is of the component class which is in the java.awt package. This method is called when the applet execution starts or the contents of applet needs to be redrawn. The paint() method takes a parameter of type Graphics class. This Graphichs object is ued to draw the text or any shapes etc. in the applet window. Its syntax is:
public void paint(Graphics g){

//code to display in the applet…
}

(4) The stop() method:

The stop() method is called when you minimize the applet window or leave the applet page and go to another page in the browser. If you are working with threads then can be restarted by the start() method. Its syntax is:

public void stop(){
//statements…
}

(5) The destroy methods:

The destroy() method is called when your applet is to be terminated and needs to removed form the memory. In this method you should do the cleanup of the objects and other resources. The destroy() method is called after the stop() method and is called only once. Its syntax is:

public void destroy(){
//code for finalization…
}

Example :-

```
//simple example of applet life cycle
import java.applet.*;
import java.awt.*;

/*  <applet code="First" width=300 height=100>
   </applet>
*/

public class First extends Applet
{
        public void init()
        {
                System.out.println(" initialization state ");
        }

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

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

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

        public void paint(Graphics g)
```

```
        {
                g.drawString(" hello how are you ?",30,40);
        }
}
```

Difference between applet and application

Application	Applet
Application is a program that can be run on user's own computer.	Applet is a program that can be run on any computer via internet.
Applications are stand-alone so it can be run independently.	Applets cannot be run independently. They are run from inside a web page using a special feature known as HTML tag.
Applications can use libraries from other language such as C or C++ through native methods	Applets are restricted from using libraries from other language.
Applications run using java interpreter.	Applets run on any browser or applet viewer.

Applet class :-

Class Applet

java.lang.Object
 |
 +--java.awt.Component
     |
     +--java.awt.Container
         |
         +--java.awt.Panel
             |
             +--**java.applet.Applet**

An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.
The Applet class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The Applet class provides a standard interface between applets and their environment.

Methods	
void	**destroy**() Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.
AccessibleContext	**getAccessibleContext**() Gets the AccessibleContext associated with this Applet.
AppletContext	**getAppletContext**() Determines this applet's context, which allows the applet to query and

	affect the environment in which it runs.
String	**getAppletInfo**() Returns information about this applet.
AudioClip	**getAudioClip**(URL url) Returns the AudioClip object specified by the URL argument.
AudioClip	**getAudioClip**(URL url, String name) Returns the AudioClip object specified by the URL and name arguments.
URL	**getCodeBase**() Gets the base URL.
URL	**getDocumentBase**() Gets the URL of the document in which this applet is embedded.
Image	**getImage**(URL url) Returns an Image object that can then be painted on the screen.
Image	**getImage**(URL url, String name) Returns an Image object that can then be painted on the screen.
Locale	**getLocale**() Gets the Locale for the applet, if it has been set.
String	**getParameter**(String name) Returns the value of the named parameter in the HTML tag.
String[][]	**getParameterInfo**() Returns information about the parameters that are understood by this applet.
void	**init**() Called by the browser or applet viewer to inform this applet that it has been loaded into the system.
boolean	**isActive**() Determines if this applet is active.
static AudioClip	**newAudioClip**(URL url) Get an audio clip from the given URL.
void	**play**(URL url) Plays the audio clip at the specified absolute URL.
void	**play**(URL url, String name) Plays the audio clip given the URL and a specifier that is relative to it.
void	**resize**(Dimension d) Requests that this applet be resized.
void	**resize**(int width, int height) Requests that this applet be resized.
void	**setStub**(AppletStub stub) Sets this applet's stub.
void	**showStatus**(String msg)

	Requests that the argument string be displayed in the "status window".
void	**start**() Called by the browser or applet viewer to inform this applet that it should start its execution.
void	**stop**() Called by the browser or applet viewer to inform this applet that it should stop its execution.

Interface AppletContext

public interface **AppletContext**

This interface corresponds to an applet's environment: the document containing the applet and the other applets in the same document.

The methods in this interface can be used by an applet to obtain information about its environment.

Method	
Applet	**getApplet**(String name) Finds and returns the applet in the document represented by this applet context with the given name.
Enumeration<Applet>	**getApplets**() Finds all the applets in the document represented by this applet context.
AudioClip	**getAudioClip**(URL url) Creates an audio clip.
Image	**getImage**(URL url) Returns an Image object that can then be painted on the screen.
InputStream	**getStream**(String key) Returns the stream to which specified key is associated within this applet context.
Iterator<String>	**getStreamKeys**() Finds all the keys of the streams in this applet context.
void	**setStream**(String key, InputStream stream) Associates the specified stream with the specified key in this applet context.
void	**showDocument**(URL url) Replaces the Web page currently being viewed with the given URL.
void	**showDocument**(URL url, String target) Requests that the browser or applet viewer show the Web page indicated by the url argument.
void	**showStatus**(String status) Requests that the argument string be displayed in the "status window".

repaint() method :-

repaint() requests an erase and redraw (*update*) after a small time delay. This allows time for more changes to the screen before actually redrawing. Without the delay you would end up repainting the screen many times a second, once for every tiny change.

void	**repaint**() Repaints this component.
void	**repaint**(int x, int y, int width, int height) Repaints the specified rectangle of this component.
void	**repaint**(long tm) Repaints the component.
void	**repaint**(long tm, int x, int y, int width, int height) Repaints the specified rectangle of this component within tm milliseconds

AWT Classes

List of AWT Classes :-

Button	Canvas	Checkbox	Choice
CheckboxGroup	Frame	Label	List
Scrollbar	TextArea	TextField	BorderLayout
CardLayout	FlowLayout	GridLayout	GridBagLayout
Graphics	Font	FontMetrics	

1] Graphics Class :-

=> This class is used for drawing several things like line, rectangle, string etc.

Constructor

Graphics

protected **Graphics**()

Constructs a new Graphics object. This constructor is the default contructor for a graphics context. Since Graphics is an abstract class, applications cannot call this constructor directly. Graphics contexts are obtained from other graphics contexts or are created by calling getGraphics on a component.

MethodS

drawRect

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

Draws the outline of the specified rectangle. The left and right edges of the rectangle are at x and x + width. The top and bottom edges are at y and y + height. The rectangle is drawn using the graphics context's current color.

Parameters:

x - the *x* coordinate of the rectangle to be drawn.
y - the *y* coordinate of the rectangle to be drawn.
width - the width of the rectangle to be drawn.
height - the height of the rectangle to be drawn.

clearRect

public abstract void **clearRect**(int x, int y, int width, int height)

Clears the specified rectangle by filling it with the background color of the current drawing surface. This operation does not use the current paint mode.

Beginning with Java 1.1, the background color of offscreen images may be system dependent. Applications should use setColor followed by fillRect to ensure that an offscreen image is cleared to a specific color.

Parameters:

x - the *x* coordinate of the rectangle to clear.
y - the *y* coordinate of the rectangle to clear.
width - the width of the rectangle to clear.
height - the height of the rectangle to clear.

drawRoundRect

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

Draws an outlined round-cornered rectangle using this graphics context's current color. The left and right edges of the rectangle are at x and x + width, respectively. The top and bottom edges of the rectangle are at y and y + height.

Parameters:

x - the *x* coordinate of the rectangle to be drawn.
y - the *y* coordinate of the rectangle to be drawn.
width - the width of the rectangle to be drawn.
height - the height of the rectangle to be drawn.
arcWidth - the horizontal diameter of the arc at the four corners.
arcHeight - the vertical diameter of the arc at the four corners.

fillRoundRect

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

Fills the specified rounded corner rectangle with the current color. The left and right edges of the rectangle are at x and x + width - 1, respectively. The top and bottom edges of the rectangle are at y and y + height - 1.

Parameters:

x - the *x* coordinate of the rectangle to be filled.
y - the *y* coordinate of the rectangle to be filled.
width - the width of the rectangle to be filled.
height - the height of the rectangle to be filled.
arcWidth - the horizontal diameter of the arc at the four corners.
arcHeight - the vertical diameter of the arc at the four corners.

drawOval

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

Draws the outline of an oval. The result is a circle or ellipse that fits within the rectangle specified by the x, y, width, and height arguments.

The oval covers an area that is width + 1 pixels wide and height + 1 pixels tall.

Parameters:

x - the *x* coordinate of the upper left corner of the oval to be drawn.
y - the *y* coordinate of the upper left corner of the oval to be drawn.
width - the width of the oval to be drawn.
height - the height of the oval to be drawn.

fillOval

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

Fills an oval bounded by the specified rectangle with the current color.

Parameters:

x - the *x* coordinate of the upper left corner of the oval to be filled.
y - the *y* coordinate of the upper left corner of the oval to be filled.

width - the width of the oval to be filled.
height - the height of the oval to be filled.

drawArc

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

Draws the outline of a circular or elliptical arc covering the specified rectangle.
The resulting arc begins at startAngle and extends for arcAngle degrees, using the current color. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.
The center of the arc is the center of the rectangle whose origin is (*x*, *y*) and whose size is specified by the width and height arguments.
The resulting arc covers an area width + 1 pixels wide by height + 1 pixels tall.
The angles are specified relative to the non-square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the bounds.

Parameters:

x - the *x* coordinate of the upper-left corner of the arc to be drawn.
y - the *y* coordinate of the upper-left corner of the arc to be drawn.
width - the width of the arc to be drawn.
height - the height of the arc to be drawn.
startAngle - the beginning angle.
arcAngle - the angular extent of the arc, relative to the start angle.

fillArc

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

Fills a circular or elliptical arc covering the specified rectangle.
The resulting arc begins at startAngle and extends for arcAngle degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation while a negative value indicates a clockwise rotation.
The center of the arc is the center of the rectangle whose origin is (*x*, *y*) and whose size is specified by the width and height arguments.

The resulting arc covers an area width + 1 pixels wide by height + 1 pixels tall.

The angles are specified relative to the non-square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the bounds.

Parameters:

x - the *x* coordinate of the upper-left corner of the arc to be filled.
y - the *y* coordinate of the upper-left corner of the arc to be filled.
width - the width of the arc to be filled.

height - the height of the arc to be filled.
startAngle - the beginning angle.
arcAngle - the angular extent of the arc, relative to the start angle.

drawPolygon

public abstract void **drawPolygon**(int[] xPoints,
int[] yPoints,
int nPoints)

Draws a closed polygon defined by arrays of *x* and *y* coordinates. Each pair of (*x*, *y*) coordinates defines a point.
This method draws the polygon defined by nPoint line segments, where the first nPoint - 1 line segments are line segments from (xPoints[i - 1], yPoints[i - 1]) to (xPoints[i], yPoints[i]), for $1 \le i \le$ nPoints. The figure is automatically closed by drawing a line connecting the final point to the first point, if those points are different.

Parameters:
xPoints - a an array of x coordinates.
yPoints - a an array of y coordinates.
nPoints - a the total number of points.

fillPolygon

public abstract void **fillPolygon**(int[] xPoints,
int[] yPoints,
int nPoints)

Fills a closed polygon defined by arrays of *x* and *y* coordinates.
This method draws the polygon defined by nPoint line segments, where the first nPoint - 1 line segments are line segments from (xPoints[i - 1], yPoints[i - 1]) to (xPoints[i], yPoints[i]), for $1 \le i \le$ nPoints. The figure is automatically closed by drawing a line connecting the final point to the first point, if those points are different.
The area inside the polygon is defined using an even-odd fill rule, also known as the alternating rule.

Parameters:
xPoints - a an array of x coordinates.
yPoints - a an array of y coordinates.
nPoints - a the total number of points.

drawString

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

Draws the text given by the specified string, using this graphics context's current font and color. The baseline of the leftmost character is at position (*x*, *y*) in this graphics context's coordinate system.

Parameters:
str - the string to be drawn.
x - the *x* coordinate.
y - the *y* coordinate.

drawChars

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

Draws the text given by the specified character array, using this graphics context's current font and color. The baseline of the first character is at position (*x*, *y*) in this graphics context's coordinate system.

Parameters:
data - the array of characters to be drawn
offset - the start offset in the data

length - the number of characters to be drawn
x - the *x* coordinate of the baseline of the text
y - the *y* coordinate of the baseline of the text

toString

public String **toString**()
Returns a String object representing this Graphics object's value.

Overrides:
toString in class Object
Returns:
a string representation of this graphics context.

drawLine

public abstract void drawLine(int x1,int y1,int x2,int y2)
Draws a line between (x1,y1) and (x2,y2) coordinates.

Parameters :
x1= left corner
y1= top corner
x2=right corner
y2=bottom corner

Class Font

The Font class allows you to specify the size and type of the font displayed in the applet window. The Font class is in the java.awt.package. To create a specific font object following constructor is used:

- Font(String name, int style, int size)

The name specifies the name of the font e.g. Arial, Verdana etc.
The style specifies the font style. It can be Font.BOLD, Font.ITALIC or Font.PLAIN. if you want your font to be both bold and italic specify its as Font.BOLD | Font.ITALIC.
The size specifies the size of the font in points.
After creating a font oobject you have to set it by the setFont() method. This method is in the component class of java.awt package. Its syntax is:

- void setFont(Font obj)

here obj is the font object that you have created.

```
//java prg for creating and setting font...
import java.applet.*;
import java.awt.*;
/*
        <applet code=fontex width=500 height=100>
        </applet>
*/
public class fontex extends Applet
{
        public void paint(Graphics g)
        {
                Font f1=new Font("Monotype Corsiva",Font.PLAIN,26);
                g.setFont(f1);
                g.drawString("This is monotype corsiva font.",10,25);
                Font f2=new Font("Arial",Font.BOLD,26);
                g.setFont(f2);
                g.drawString("This is Arial- Bold font.",10,55);
                Font f3=new Font("Times new roman",Font.BOLD|Font.ITALIC,26);
                g.setFont(f3);
```

```
            g.drawString("This is times new roman -bold and italic font.",10,95);
        }
}
```

16] Class FontMetrics

The FontMetrics class is used to get various measures of the font. All the fonts do not hava the same dimension. Thus we have to manage the text output. The FontMetrics class can be helpful to manage the text output such as the distance between two lines etc. the following terms are used to describe fonts:

- Baseline:

It is the line to which the business the bottoms of characters are aligned to.

- Ascent:

It is the distance from the baseline to the top of a character.

- Descent:

It is distance from the baseline to the bottom of a character.

- Height:

It is the height of the tallest character in the font.

- Leading:

It is distance between two lines i.e. the distance between bottom of one line and top of the next line characters.

Method	
int	**bytesWidth**(byte[] data, int off, int len) Returns the total advance width for showing the specified array of bytes in this Font.
int	**charsWidth**(char[] data, int off, int len) Returns the total advance width for showing the specified array of characters in this Font.
int	**charWidth**(char ch) Returns the advance width of the specified character in this Font.
int	**charWidth**(int ch) Returns the advance width of the specified character in this Font.
int	**getAscent**() Gets the font ascent.
int	**getDescent**() Gets the font descent.
Font	**getFont**() Gets the font.
int	**getHeight**() Gets the standard height of a line of text in this font.
int	**getLeading**() Gets the standard leading, or line spacing, for the font.
int	**getMaxAdvance**() Gets the maximum advance width of any character in this Font.
int	**getMaxAscent**() Gets the maximum ascent of all characters in this Font.
int	**getMaxDecent**() For backward compatibility only.
int	**getMaxDescent**() Gets the maximum descent of all characters in this Font.
int[]	**getWidths**() Gets the advance widths of the first 256 characters in the Font.
int	**stringWidth**(String str) Returns the total advance width for showing the specified String in this Font.

String	**toString**() Returns the String representation of this FontMetric's values.

```
//java prg for showing the use of FontMetrics class...
import java.applet.*;
import java.awt.*;
/*
        <applet code=fontmetricsex width=200 height=100>
        </applet>
*/
public class fontmetricsex extends Applet
{
        public void paint(Graphics g)
        {
                Dimension dim=getSize();
                g.setColor(Color.red);
                int a,d,w,h,s;
                FontMetrics fm=g.getFontMetrics();
                a=fm.getAscent();
                d=fm.getDescent();
                w=dim.width;
                h=dim.height;
                s=fm.stringWidth("Welcome");
                int x=(w-s)/2;
                int y=a+(h-(a+d))/2;
                g.drawString("Welcome",x,y);
                g.drawRect(0,0,w-1,h-1);
        }
}
```

LAYOUT MANAGERS

- A layout manager automatically arranges your controls within a window. Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface.
- It can be set by **setLayout()** method. If the method is not called then default layout is used. Whenever a container is resized, the layout manager is used to position each of the components within it.
  void setLayout(LayoutManager layoutobj)
- here layoutobj is a reference to the desired layout manager. If we want to disable the layout manager and position components manually, we have to pass **null** for layoutobj.
- Each of managers keeps track of a list of components that are stored by their names. Whenever new control is added that is notified by the manager. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize()** and **preferredLayoutSize()** methods. Each components managed by manager contains **getPreferredSize()** and **getMinimumSize().**

(A) FlowLayout:

- It is the default layout. It implements a simple layout style, which is similar to word flow in text editor. Components are laid out from left to right and top to down. When new components is not fitted to the current line then placed in new line. A small space is left between each component.

Constructor: **FlowLayout()**
FlowLayout(int how**)**
FlowLayout(int how**, int** horz**, int** vert**)**
how FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER
Horz, vert horizontal and vertical space between each component)
Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code="LayOut1" width=300 height=300> </applet>*/
public class LayOut1 extends Applet implements ItemListener
{
String msg="";
Checkbox c1,c2;
public void init()
{
setLayout(new FlowLayout(FlowLayout.CENTER));
c1 = new Checkbox("first",null,true);
c2 = new Checkbox("second");
add(c1);
add(c2);
c1.addItemListener(this);
c2.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{ repaint(); }
public void paint(Graphics g)
{
msg = "Current state : ";
g.drawString(msg,6,60);
msg = "first : " + c1.getState();
g.drawString(msg,6,80);
msg = "second : " + c2.getState();
g.drawString(msg,6,100);
}
}
```

(B) BorderLayout:

- It implements a common layout style for top-level window. It has 4 narrow, fixed-width components at the edges and one large area in the center. For edges are called north, south, east, west. Middle area is called center.

Constructor: **BordetLayout()**

BordetLayout(int horz**, int** vert**)**

The BordetLayout defines the following constants that specifies the regions.
BordetLayout.CENTER BordetLayout.NORTH BordetLayout.SOUTH
BordetLayout.EAST BordetLayout.WEST

At the time of adding components, following form of add() is used.

void **add(Component** obj**, object** region**)**

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code="LayOut2" width=300 height=300> </applet>*/
public class LayOut2 extends Applet
{
public void init()
{
setLayout(new BorderLayout());
```

```
add(new Button("top"),BorderLayout.NORTH);
add(new Button("bottom"),BorderLayout.SOUTH);
add(new Button("right"),BorderLayout.EAST);
add(new Button("left"),BorderLayout.WEST);
String msg = "This is the center\n";
msg += "Part of the border layout.\n";
msg += "This part is big according to the others.";
add(new TextArea(msg,10,10),BorderLayout.CENTER);
}
}
```

(C) GridLayout:

- It layout components in 2-D grid. When we instantiate a GridLayout, we have to define the number of rows and columns.

Constructor: **GridLayout()**

GridLayout(int numrows**, int** numcolumns**)**

GridLayout(int numrows**, int** numcolumns**, int** horz**, int** vert**)**

Example:

```
import java.awt.*;
import java.applet.*;
/*<applet code="LayOut3" width=300 height=300> </applet>*/
public class LayOut3 extends Applet
{
static final int n = 4;
public void init()
{
setLayout(new GridLayout(n,n));
setFont(new Font("Garamond",Font.BOLD,24));
for(int i=0;i<n;i++)
{
for (int j=0;j<n;j++)
{
int k = i*n+j;
if(k>0)
add(new Button(""+k));
}
}
}
}
```

(D) CardLayout:

- It is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled. It can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.

Constructor: **CardLayout()**

CardLayout(int horz**, int** vert**)**

- Use of cardlayout requires a bitmore work than other layouts. These card are held in an object of type **Panel**, which must have CardLayout selected as its layout manager. So we must create a panel that contains the deck and a panel for each card in the deck.
- Then we have to add appropriate component of card to each panel. Then we have to add panel to main applet panel. To add card to panel we have to use add().

void **add(Component** panelobj**, Object** name**)**

- After creation of deck, program activates a card by calling one of the following methods defined by **CardLayout**.

void **first(Container** deck**)**
void **last(Container** deck**)**
void **next(Container** deck**)**
void **previous(Container** deck**)**
void **show(Container** deck**, String** cardname**)**

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code="LayOut4" width=300 height=300> </applet>*/
public class LayOut4 extends Applet implements ActionListener, MouseListener
{
Checkbox c1,c2,c3,c4;
Panel p1;
CardLayout cl1;
Button b1,b2;
public void init()
{
b1 = new Button("First button");
b2 = new Button("Second button");
add(b1);
add(b2);
cl1 = new CardLayout();
p1 = new Panel();
p1.setLayout(cl1);
c1 = new Checkbox("First checkbox",null,true);
c2 = new Checkbox("Second checkbox");
c3 = new Checkbox("Third checkbox");
c4 = new Checkbox("Fourth checkbox");
Panel p2 = new Panel();
p2.add(c1);
p2.add(c2);
Panel p3 = new Panel();
p3.add(c3);
p3.add(c4);
p1.add(p2,"first panel");
p1.add(p3,"second panel");
add(p1);
b1.addActionListener(this);
b2.addActionListener(this);
addMouseListener(this);
}
public void mousePressed(MouseEvent me)
{
cl1.next(p1);
}
// PROVIDE EMPTY IMPLEMENTATIONS FOR THE
// OTHER MOUSELITENER METHODS
public void mouseClicked(MouseEvent me){}
```

```
public void mouseEntered(MouseEvent me){}
public void mouseExited(MouseEvent me){}
public void mouseReleased(MouseEvent me){}
public void actionPerformed(ActionEvent ae)
{
if(ae.getSource() == b1)
cl1.show(p1,"first panel");
else
cl1.show(p1,"second panel");
}
}
```

(E) GridBag Layout:

- The last of the layout managers available through Java is grid bag layout.
- A complex extension of the grid layout manager. A grid bag layout differs from grid layout in the following ways:
  - A component can take up more than one cell in the grid.
  - The proportions between different rows and columns do not have to be equal.
  - A component does not have to fill the entire cell (or cells) that it occupies.
  - A component can be aligned along any edge of a cell.
- A grid bag layout requires the GridBagLayout and GridBagConstraints classes, which both are part of the java.awt package. GridBagLayout is the layout manager, and GridBagConstraints defines the placement of components in the grid.
- The constructor for the grid bag layout manager takes no arguments and can be applied to a container like any other manager. The following statements could be used in a frame or window's constructor method to use grid bag layout in that container:

```
Container pane = getContentPane();
GridBagLayout bag = new GridBagLayout();
pane.setLayout(bag);
```

- In a grid bag layout, each component uses a GridBagConstraints object to dictate the cell or cells that it occupies in the grid, its size, and other aspects of its presentation. A GridBagConstraints object has 11 instance variables that determine component placement:
  - gridx—The x position of the cell that holds the component (if it spans several cells, the x position of the upper-left portion of the component)
  - gridy—The y position of the cell or its upper-left portion
  - gridwidth—The number of cells the component occupies in a horizontal direction
  - gridheight—The number of cells the component occupies in a vertical direction
  - weightx—A value that indicates the component's size relative to other components on the same row of the grid
  - weighty—A value that indicates its size relative to components on the same grid column
  - anchor—A value that determines where the component is displayed within its cell (if it doesn't fill the entire cell)
  - fill—A value that determines whether the component expands horizontally or vertically to fill its cell
  - insets—An Insets object that sets the whitespace around the component inside its cell
  - ipadx—The amount to expand the component's width beyond its minimum size
  - ipady—The amount to expand the component's height
- With the exception of insets, all these can hold integer values. The easiest way to use this class is to create a constraints object with no arguments and set its variables individ ually. Variables not explicitly set use their default values. The following code creates a grid bag layout and a constraints object used to place components in the grid:

```
Container pane = getContentPane();
```

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
pane.setLayout(gridbag);
```

- A component is added to a grid bag layout in two steps:

1. The layout manager's setConstraints(Component, GridBagConstraints) method is called with the component and constraints objects as arguments.
2. The component is added to a container that uses that manager.

Example:

```
import java.applet.*;
import java.awt.*;
/*
<applet code="GridBag" width=300 height=300> </applet>
*/
public class GridBag extends Applet
{
String str="";
public void init()
{
GridBagConstraints gbc = new GridBagConstraints();
GridBagLayout gb = new GridBagLayout();
setLayout(gb);
Checkbox c1 = new Checkbox("First");
Checkbox c2 = new Checkbox("Second");
Checkbox c3 = new Checkbox("Third");
Checkbox c4 = new Checkbox("Fourth");
gbc.gridwidth = GridBagConstraints.REMAINDER;
gb.setConstraints(c1,gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gb.setConstraints(c2,gbc);
gbc.weighty=0.2;
gbc.weightx = 0.5;
gbc.gridwidth = GridBagConstraints.REMAINDER;
gb.setConstraints(c3,gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gb.setConstraints(c4,gbc);
add(c1);
add(c2);
add(c3);
add(c4);
}
}
```

BoxLayout

The BoxLayout is a general purpose layout manager which is an extension of FlowLayout. It places the components on top of each other or places them one after another in a row. Its constructor is:

- BoxLayout(Container obj, int placement)

Here, the obj is the object of a container such as a panel and the placement can be one of the following:

- BoxLayout.PAGE_AXIS: It places the component like a stack i.e. on top of each other.
- BoxLayout.LINE_AXIS : it places the components in a line, i.e. one after another.

The following example demonstrates the BoxLayout:

```
import javax.swing.*;
import java.awt.*;
```

```
/*
<applet code=boxlayoutex width=250 height=200>
</applet>
*/
public class boxlayoutex extends JApplet{
public void init()
{
        JPanel pane1=new JPanel();
        JPanel pane2=new JPanel();
        pane1.setLayout(new BoxLayout(pane1,BoxLayout.PAGE_AXIS));
        JButton b1=new JButton("Button1");
        JButton b2=new JButton("Button2");
        JButton b3=new JButton("Button3");
        pane1.add(b1);
        pane1.add(b2);
        pane1.add(b3);
        pane2.setLayout(new BoxLayout(pane2,BoxLayout.LINE_AXIS));
        pane2.add(Box.createRigidArea(new Dimension(50,50)));
        JButton b4=new JButton("Button4");
        JButton b5=new JButton("Button5");
        JButton b6=new JButton("Button6");
        pane2.add(b4);
        pane2.add(b5);
        pane2.add(b6);
        Container pane=getContentPane();
        pane.add(pane1,BorderLayout.NORTH);
        pane.add(pane2,BorderLayout.EAST);
}
}
```

SpringLayout

The SpringLayout is a very flexible layout that has many features for specifying the size of the components. You can have different size rows and/or columns in a container. In SpringLayout you can define a fixed distance between the left edge of one component and right edge of another component. The constructor for SpringLayout is:

- StringLayout()
  You can put constraints of SpringLayout by the following methods:
- void putConstraint(String edge1, Component obj1, int distance, String edge2, Component obj2)
- void putConstraint(String edge1, Component obj1, Spring distance, String edge2, Component obj2)

here, the edge can be one of the following:

- SpringLayout.NORTH
- SpringLayout.SOUTH
- SprintLayout.EAST
- SpringLayout.WEST

And obj1 and obj2 are the components between whom the distance is specified.
The following example shows the SpringLayout demo:

```
//java prg for spring layout....
import java.awt.*;
import javax.swing.*;
/*
        <applet code=springlayoutex width=250 height=100>
        </applet>
```

```
*/
public class springlayoutex extends JApplet
{
	public void init()
	{
		Container pane=getContentPane();
		SpringLayout spring=new SpringLayout();
		pane.setLayout(spring);
		JLabel l1,l2;
		JTextField t1,t2;
		l1=new JLabel("Name : ");
		l2=new JLabel("E-mail : ");
		t1=new JTextField(10);
		t2=new JTextField(15);
		pane.add(l1);
		pane.add(t1);
		pane.add(l2);
		pane.add(t2);
		spring.putConstraint(SpringLayout.WEST,l1,10,SpringLayout.WEST,pane);
		spring.putConstraint(SpringLayout.NORTH,l1,10,SpringLayout.NORTH,pane);
		spring.putConstraint(SpringLayout.WEST,t1,10,SpringLayout.EAST,l1);
		spring.putConstraint(SpringLayout.NORTH,t1,10,SpringLayout.NORTH,pane);
		spring.putConstraint(SpringLayout.WEST,l2,10,SpringLayout.WEST,pane);
		spring.putConstraint(SpringLayout.NORTH,l2,30,SpringLayout.NORTH,l1);
		spring.putConstraint(SpringLayout.WEST,t2,10,SpringLayout.EAST,l2);
		spring.putConstraint(SpringLayout.NORTH,t2,40,SpringLayout.NORTH,pane);
	}
}
```

Using No Layout Manager

To absolute positioning components, you can set No Layout to your container. Absolute position means you specify the location of your component using setBounds() method.

Setting no layout is very easy. You have to just follow the below steps:

(1) Set the container's layout manager to null [call setLayout(null)]
(2) Set the bounds of the component.[call setBounds()].

Following is an example which has only two components: namely a label and a text field. We have set bounds for these two components.

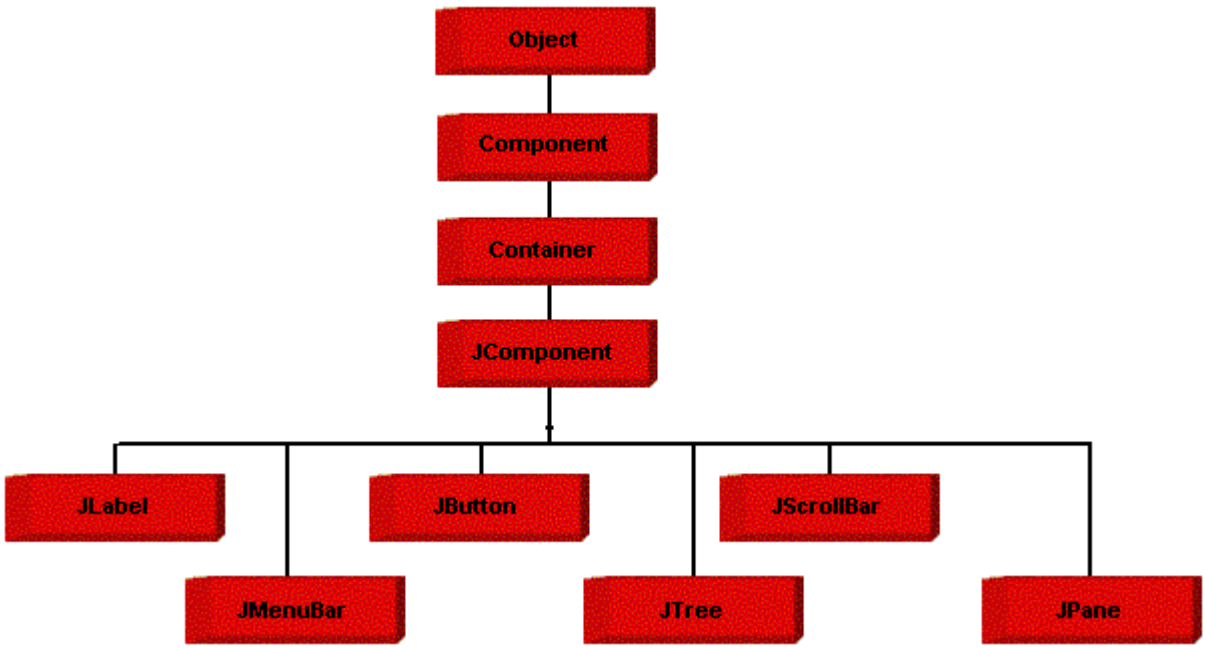
Example

```
// A simple example of No Layout Manager.
import javax.swing.*;
public class nolayoutex extends JFrame
{
	public  static void main(String args[])
	{
		JFrame f=new JFrame("no layout Example");
		f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		f.setLayout(null);//setting no layout
		JLabel  lbl=new JLabel("enter Name:");
		lbl.setBounds(25, 25, 90, 25);
		JTextField txt= new JTextField();
		txt.setBounds(120, 25, 100, 25);
		f.add(lbl);
		f.add(txt);
		f.setSize(300, 100);
		f.setvisible(true);
	}
}
```

UNIT-5

GUI Using SWING And Event Handling

- Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
- It requires to import
  javax.swing.*;



Difference between Awt and Swing Components

AWT	SWING
AWT components are dependent on the nature of the platform.	Java swing components are not dependent on the nature of the platform. They are purely scripted in Java.
Heavyweight in character.	Mostly lightweight in character
AWT components are less in number in comparison to Swing components.	Java Swing components are very powerful and much more in number. They are represented by lists, scroll panes, tables, tabbed panes, colour choosers, etc.
The AWT full form is Abstract windows toolkit.	Swing components in Java are also referred to as JFC's (Java Foundation classes).
AWT components need the java.awt package.	Swing components in Java need the javax.swing package.
AWT components require and occupy larger memory space.	Swing components do not occupy as much memory space as awt components.

Swing Classes :-

- AbstractButton
- ButtonGroup
- ImageIcon
- JApplet
- JButton
- JCheckBox
- JComboBox
- JList

- JLabel
- JRadioButton
- JScrollBar
- JPanel
- JFrame
- JTextField
- JPasswordField
- JTextArea
- JMenu
- JMenuBar
- JMenuItem

JApplet :-

The JApplet class is slightly incompatible with java.applet.Applet. JApplet contains a JRootPane as it's only child. The **contentPane** should be the parent of any children of the JApplet. This is different than java.applet.Applet, e.g. to add a child to an an java.applet.Applet you'd write:

```
applet.add(child);
```

Constructor	
JApplet() Creates a swing applet instance.	
Method	
Container	**getContentPane**() Returns the contentPane object for this applet.
Component	**getGlassPane**() Returns the glassPane object for this applet.
JMenuBar	**getJMenuBar**() Returns the menubar set on this applet.
JLayeredPane	**getLayeredPane()** **Returns the layeredPane object for this applet.**
JRootPane	**getRootPane()** **Returns the rootPane object for this applet.**
protected boolean	**isRootPaneCheckingEnabled()**
protected String	**paramString()** **Returns a string representation of this JApplet.**
void	**remove(Component comp)** **Removes the specified component from this container.**
void	**setContentPane(Container contentPane)** **Sets the contentPane property.**
void	**setGlassPane(Component glassPane)** **Sets the glassPane property.**
void	**setJMenuBar(JMenuBar menuBar)**

	Sets the menubar for this applet.
void	**setLayeredPane(JLayeredPane layeredPane)** **Sets the layeredPane property.**
void	**setLayout(LayoutManager manager)** **By default the layout of this component may not be set, the layout of its contentPane should be set instead.**
protected void	**setRootPane(JRootPane root)** **Sets the rootPane property.**
protected void	**setRootPaneCheckingEnabled(boolean enabled)** **If true then calls to add() and setLayout() will cause an exception to be thrown.**
void	**update(Graphics g)** **Just calls paint(g).**

JLabel :-

A JLabel object provides text instructions or information on a GUI — display a single line of read-only text, an image or both text and image.

We use a Swing JLabel when we need a user interface component that displays a message or an image.

Provide text instructions on a GUI

Read-only text

Programs rarely change a label's contents

Class **JLabel** (subclass of **JComponent**)

Constructor
JLabel() Creates a JLabel instance with no image and with an empty string for the title.
JLabel(Icon image) Creates a JLabel instance with the specified image.
JLabel(Icon image, int horizontalAlignment) Creates a JLabel instance with the specified image and horizontal alignment.
JLabel(String text) Creates a JLabel instance with the specified text.
JLabel(String text, Icon icon, int horizontalAlignment) Creates a JLabel instance with the specified text, image, and horizontal alignment.
JLabel(String text, int horizontalAlignment) Creates a JLabel instance with the specified text and horizontal alignment.

Method	
Icon	**getDisabledIcon**() Returns the value of the disabledIcon property if it's been set, If it hasn't been set and the value of the icon property is an ImageIcon, we compute a "grayed out" version of the icon and update the disabledIcon property with that.
int	**getDisplayedMnemonic**() Return the keycode that indicates a mnemonic key.

int	**getDisplayedMnemonicIndex**() Returns the character, as an index, that the look and feel should provide decoration for as representing the mnemonic character.
int	**getHorizontalAlignment**() Returns the alignment of the label's contents along the X axis.
int	**getHorizontalTextPosition**() Returns the horizontal position of the label's text, relative to its image.
Icon	**getIcon**() Returns the graphic image (glyph, icon) that the label displays.
int	**getIconTextGap**() Returns the amount of space between the text and the icon displayed in this label.
String	**getText**() Returns the text string that the label displays.
int	**getVerticalAlignment**() Returns the alignment of the label's contents along the Y axis.
int	**getVerticalTextPosition**() Returns the vertical position of the label's text, relative to its image.
void	**setDisabledIcon**(Icon disabledIcon) Set the icon to be displayed if this JLabel is "disabled" (JLabel.setEnabled(false)).
void	**setDisplayedMnemonic**(char aChar) Specifies the displayedMnemonic as a char value.
void	**setDisplayedMnemonic**(int key) Specify a keycode that indicates a mnemonic key.
void	**setDisplayedMnemonicIndex**(int index) Provides a hint to the look and feel as to which character in the text should be decorated to represent the mnemonic.
void	**setHorizontalAlignment**(int alignment) Sets the alignment of the label's contents along the X axis.
void	**setHorizontalTextPosition**(int textPosition) Sets the horizontal position of the label's text, relative to its image.
void	**setIcon**(Icon icon) Defines the icon this component will display.
void	**setIconTextGap**(int iconTextGap) If both the icon and text properties are set, this property defines the space between them.
void	**setText**(String text) Defines the single line of text this component will display.
void	**setVerticalAlignment**(int alignment) Sets the alignment of the label's contents along the Y axis.

void	**setVerticalTextPosition**(int textPosition) Sets the vertical position of the label's text, relative to its image.

ImageIcon :-

- Constructors :-

  ImageIcon(String filename)
  ImageIcon(URL url)

It implements Icon interface which has following methods :

- Methods:-

  int getIconHeight()
  int getIconWidth()
  void paintIcon(Component comp, Graphics g, int x, int y)

JTextField :-

JTextField allows editing/displaying of a single line of text. New features include the ability to justify the text left, right, or center, and to set the text's font. When the user types data into them and presses the Enter key, an action event occurs.

Constructor
JTextField() Constructs a new TextField.
JTextField(int columns) Constructs a new empty TextField with the specified number of columns.
JTextField(String text) Constructs a new TextField initialized with the specified text.
JTextField(String text, int columns) Constructs a new TextField initialized with the specified text and columns.

Method	
void	**addActionListener**(ActionListener l) Adds the specified action listener to receive action events from this textfield.
Action	**getAction**() Returns the currently set Action for this ActionEvent source, or null if no Action is set.
ActionListener[]	**getActionListeners**() Returns an array of all the ActionListeners added to this JTextField with addActionListener().
Action[]	**getActions**() Fetches the command list for the editor.
int	**getColumns**() Returns the number of columns in this TextField.
protected int	**getColumnWidth**() Returns the column width.
int	**getHorizontalAlignment**() Returns the horizontal alignment of the text.

void	**removeActionListener**(ActionListener l) Removes the specified action listener so that it no longer receives action events from this textfield.
void	**setAction**(Action a) Sets the Action for the ActionEvent source.
void	**setActionCommand**(String command) Sets the command string used for action events.
void	**setColumns**(int columns) Sets the number of columns in this TextField, and then invalidate the layout.
void	**setFont**(Font f) Sets the current font.
void	**setHorizontalAlignment**(int alignment) Sets the horizontal alignment of the text.

JPasswordField

JPasswordField (a direct subclass of JTextField) you can suppress the display of input. Each character entered can be replaced by an echo character. This allows confidential input for passwords, for example. By default, the echo character is the asterisk, *. When the user types data into them and presses the Enter key, an action event occurs.

Constructor
JPasswordField() Constructs a new JPasswordField, with a default document, null starting text string, and 0 column width.
JPasswordField(Document doc, String txt, int columns) Constructs a new JPasswordField that uses the given text storage model and the given number of columns.
JPasswordField(int columns) Constructs a new empty JPasswordField with the specified number of columns.
JPasswordField(String text) Constructs a new JPasswordField initialized with the specified text.
JPasswordField(String text, int columns) Constructs a new JPasswordField initialized with the specified text and columns.

Method	
boolean	**echoCharIsSet**() Returns true if this JPasswordField has a character set for echoing.
char	**getEchoChar**() Returns the character to be used for echoing.
char[]	**getPassword**() Returns the text contained in this TextComponent.
String	**getText**()

	Deprecated. *As of Java 2 platform v1.2, replaced by getPassword.*
void	**setEchoChar**(char c) Sets the echo character for this JPasswordField.

JTextArea

JTextArea allows editing of multiple lines of text. JTextArea can be used in conjunction with class JScrollPane to achieve scrolling. The underlying JScrollPane can be forced to always or never have either the vertical or horizontal scrollbar.

Constructor
JTextArea() Constructs a new TextArea.
JTextArea(Document doc) Constructs a new JTextArea with the given document model, and defaults for all of the other arguments (null, 0, 0).
JTextArea(Document doc, String text, int rows, int columns) Constructs a new JTextArea with the specified number of rows and columns, and the given model.
JTextArea(int rows, int columns) Constructs a new empty TextArea with the specified number of rows and columns.
JTextArea(String text) Constructs a new TextArea with the specified text displayed.
JTextArea(String text, int rows, int columns) Constructs a new TextArea with the specified text and number of rows and columns.

Method	
void	**append**(String str) Appends the given text to the end of the document.
int	**getColumns**() Returns the number of columns in the TextArea.
protected int	**getColumnWidth**() Gets column width.
int	**getLineCount**() Determines the number of lines contained in the area.
boolean	**getLineWrap**() Gets the line-wrapping policy of the text area.
int	**getRows**() Returns the number of rows in the TextArea.
int	**getTabSize**() Gets the number of characters used to expand tabs.
void	**insert**(String str, int pos) Inserts the specified text at the specified position.

void	**replaceRange**(String str, int start, int end) Replaces text from the indicated start to end position with the new text specified.
void	**setColumns**(int columns) Sets the number of columns for this TextArea.
void	**setFont**(Font f) Sets the current font.
void	**setLineWrap**(boolean wrap) Sets the line-wrapping policy of the text area.
void	**setRows**(int rows) Sets the number of rows for this TextArea.
void	**setTabSize**(int size) Sets the number of characters to expand tabs to.

JRadioButton

A set of radio buttons can be associated as a group in which only one button at a time can be selected.

Constructor
JRadioButton() Creates an initially unselected radio button with no set text.
JRadioButton(Icon icon) Creates an initially unselected radio button with the specified image but no text.
JRadioButton(Icon icon, boolean selected) Creates a radio button with the specified image and selection state, but no text.
JRadioButton(String text) Creates an unselected radio button with the specified text.
JRadioButton(String text, boolean selected) Creates a radio button with the specified text and selection state.
JRadioButton(String text, Icon icon) Creates a radio button that has the specified text and image, and that is initially unselected.
JRadioButton(String text, Icon icon, boolean selected) Creates a radio button that has the specified text, image, and selection state.

JCheckBox

JCheckBox is not a member of a checkbox group. A checkbox can be selected and deselected, and it also displays its current state. It is used for multiple selection.

Constructor
JCheckBox() Creates an initially unselected check box button with no text, no icon.
JCheckBox(Icon icon) Creates an initially unselected check box with an icon.
JCheckBox(Icon icon, boolean selected)

Creates a check box with an icon and specifies whether or not it is initially selected.
JCheckBox(String text) Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected) Creates a check box with text and specifies whether or not it is initially selected.
JCheckBox(String text, Icon icon) Creates an initially unselected check box with the specified text and icon.
JCheckBox(String text, Icon icon, boolean selected) Creates a check box with text and icon, and specifies whether or not it is initially selected.

Method	
protected void	setSelected(boolean state) The state of the check box can be changed.

JComboBox

JComboBox is like a drop down box — you can click a drop-down arrow and select an option from a list. It generates ItemEvent. For example, when the component has focus, pressing a key that corresponds to the first character in some entry's name selects that entry. A vertical scrollbar is used for longer lists.

Constructors :-

JComboBox()

JCombobox(Vector v)

void	**addItem**(Object anObject) Adds an item to the item list.
String	**getActionCommand**() Returns the action command that is included in the event sent to action listeners.
Object	**getItemAt**(int index) Returns the list item at the specified index.
int	**getItemCount**() Returns the number of items in the list.
int	**getMaximumRowCount**() Returns the maximum number of items the combo box can display without a scrollbar
int	**getSelectedIndex**() Returns the first item in the list that matches the given item.
Object	**getSelectedItem**() Returns the current selected item.
void	**insertItemAt**(Object anObject, int index) Inserts an item into the item list at a given index.
boolean	**isEditable**()

	Returns true if the JComboBox is editable.
void	**removeAllItems**() Removes all items from the item list.
void	**removeItem**(Object anObject) Removes an item from the item list.
void	**removeItemAt**(int anIndex) Removes the item at anIndex This method works only if the JComboBox uses a mutable data mod
protected void	**selectedItemChanged**() This protected method is implementation specific.
void	**setActionCommand**(String aCommand) Sets the action command that should be included in the event sent to action listeners.
void	**setEditable**(boolean aFlag) Determines whether the JComboBox field is editable.
void	**setEnabled**(boolean b) Enables the combo box so that items can be selected.
void	**setMaximumRowCount**(int count) Sets the maximum number of rows the JComboBox displays.
void	**setSelectedIndex**(int anIndex) Selects the item at index anIndex.
void	**setSelectedItem**(Object anObject) Sets the selected item in the combo box display area to the object in the argument.

Java JList class example

JList

JList provides a scrollable set of items from which one or more may be selected. JList can be populated from an Array or Vector. JList does not support scrolling directly—instead, the list must be associated with a scrollpane. The view port used by the scrollpane can also have a user-defined border.

Constructor Summary
JList() Constructs a JList with an empty model.
JList(Vector listData) Constructs a JList that displays the elements in the specified Vector.

Method Summary	
boolean	**getDragEnabled**() Gets the dragEnabled property.
int	**getFirstVisibleIndex**() Returns the index of the first visible cell.
int	**getFixedCellHeight**()

	Returns the fixed cell height value -- the value specified by setting the fixedCellHeight property, rather than that calculated from the list elements.
int	**getFixedCellWidth**() Returns the fixed cell width value -- the value specified by setting the fixedCellWidth property, rather than that calculated from the list elements.
int	**getLastVisibleIndex**() Returns the index of the last visible cell.
int	**getLeadSelectionIndex**() Returns the second index argument from the most recent addSelectionInterval or setSelectionInterval call.

JFrame

The components added to the frame are referred to as its contents; these are managed by the contentPane. To add a component to a JFrame, we must use its contentPane instead.JFrame is a Window with border, title and buttons. When JFrame is set visible, an event dispatching thread is started. JFrame objects store several objects including a Container object known as the content pane. To add a component to a JFrame, add it to the content pane.

Constructor	
JFrame() Constructs a new frame that is initially invisible.	
JFrame(String title) Creates a new, initially invisible Frame with the specified title.	

Method Summary	
Container	getContentPane() Returns the contentPane object for this frame.
int	getDefaultCloseOperation() Returns the operation that occurs when the user initiates a "close" on this frame.
void	remove(Component comp) Removes the specified component from this container.
void	setContentPane(Container contentPane) Sets the contentPane property.
void	setDefaultCloseOperation(int operation) Sets the operation that will happen by default when the user initiates a "close" on this frame.
void	setLayout(LayoutManager manager) By default the layout of this component may not be set, the layout of its contentPane should be set instead.

JMenu

Swing provides support for pull-down and popup menus. A JMenubar can contain several JMenu 's. Each of the JMenu 's can contain a series of JMenuItem 's that you can select.

How Menu's Are Created?

1. First, A JMenubar is created
2. Then, we attach all of the menus to this JMenubar.
3. Then we add JMenuItem 's to the JMenu 's.
4. The JMenubar is then added to the frame. By default, each JMenuItem added to a JMenu is enabled—that is, it can be selected. In certain situations, we may need to disable a JMenuItem. This is done by calling setEnabled(). The setEnabled()
method also allows components to be enabled.

Constructor
JMenu() Constructs a new JMenu with no text.
JMenu(String s) Constructs a new JMenu with the supplied string as its text.
JMenu(String s, boolean b) Constructs a new JMenu with the supplied string as its text and specified as a tear-off menu or not.

Method	
Component	**add**(Component c) Appends a component to the end of this menu.
Component	**add**(Component c, int index) Adds the specified component to this container at the given position.
JMenuItem	**add**(JMenuItem menuItem) Appends a menu item to the end of this menu.
JMenuItem	**add**(String s) Creates a new menu item with the specified text and appends it to the end of this menu.
void	**addSeparator**() Appends a new separator to the end of the menu.
void	**doClick**(int pressTime) Programmatically performs a "click".
Component	**getComponent**() Returns the java.awt.Component used to paint this MenuElement.
int	**getDelay**() Returns the suggested delay, in milliseconds, before submenus are popped up or down.
JMenuItem	**getItem**(int pos) Returns the JMenuItem at the specified position.
int	**getItemCount**() Returns the number of items on the menu, including separators.
Component	**getMenuComponent**(int n) Returns the component at position n.

int	**getMenuComponentCount**() Returns the number of components on the menu.
Component[]	**getMenuComponents**() Returns an array of Components of the menu's subcomponents.
MenuListener[]	**getMenuListeners**() Returns an array of all the MenuListeners added to this JMenu with addMenuListener().
JPopupMenu	**getPopupMenu**() Returns the popupmenu associated with this menu.
protected Point	**getPopupMenuOrigin**() Computes the origin for the JMenu's popup menu.
MenuElement[]	**getSubElements**() Returns an array of MenuElements containing the submenu for this menu component.
JMenuItem	**insert**(JMenuItem mi, int pos) Inserts the specified JMenuitem at a given position.
void	**insert**(String s, int pos) Inserts a new menu item with the specified text at a given position.
void	**insertSeparator**(int index) Inserts a separator at the specified position.
boolean	**isMenuComponent**(Component c) Returns true if the specified component exists in the submenu hierarchy.
boolean	**isPopupMenuVisible**() Returns true if the menu's popup window is visible.
boolean	**isSelected**() Returns true if the menu is currently selected (highlighted).
boolean	**isTearOff**() Returns true if the menu can be torn off.
boolean	**isTopLevelMenu**() Returns true if the menu is a 'top-level menu', that is, if it is the direct child of a menubar.
void	**menuSelectionChanged**(boolean isIncluded) Messaged when the menubar selection changes to activate or deactivate this menu.
void	**remove**(Component c) Removes the component c from this menu.
void	**remove**(int pos) Removes the menu item at the specified index from this menu.
void	**remove**(JMenuItem item) Removes the specified menu item from this menu.

void	**removeAll**() Removes all menu items from this menu.
void	**setDelay**(int d) Sets the suggested delay before the menu's PopupMenu is popped up or down.
void	**setMenuLocation**(int x, int y) Sets the location of the popup component.
void	**setPopupMenuVisible**(boolean b) Sets the visibility of the menu's popup.
void	**setSelected**(boolean b) Sets the selection status of the menu.

Class JMenuBar

An implementation of a menu bar. You add JMenu objects to the menu bar to construct a menu. When the user selects a JMenu object, its associated JPopupMenu is displayed, allowing the user to select one of the JMenuItems on it.

Constructor
JMenuBar() Creates a new menu bar.

Method	
JMenu	**add**(JMenu c) Appends the specified menu to the end of the menu bar.
void	**addNotify**() Overrides JComponent.addNotify to register this menu bar with the current keyboard manager.
Component	**getComponent**() Implemented to be a MenuElement.
Component	**getComponentAtIndex**(int i) **Deprecated.** *replaced by getComponent(int i)*
int	**getComponentIndex**(Component c) Returns the index of the specified component.
JMenu	**getHelpMenu**() Gets the help menu for the menu bar.
Insets	**getMargin**() Returns the margin between the menubar's border and its menus.
JMenu	**getMenu**(int index) Returns the menu at the specified position in the menu bar.

int	**getMenuCount**() Returns the number of items in the menu bar.
MenuElement[]	**getSubElements**() Implemented to be a MenuElement -- returns the menus in this menu bar.
boolean	**isBorderPainted**() Returns true if the menu bars border should be painted.
boolean	**isSelected**() Returns true if the menu bar currently has a component selected.
void	**menuSelectionChanged**(boolean isIncluded) Implemented to be a MenuElement -- does nothing.
protected void	**paintBorder**(Graphics g) Paints the menubar's border if BorderPainted property is true.
void	**removeNotify**() Overrides JComponent.removeNotify to unregister this menu bar with the current keyboard manager.
void	**setBorderPainted**(boolean b) Sets whether the border should be painted.
void	**setHelpMenu**(JMenu menu) Sets the help menu that appears when the user selects the "help" option in the menu bar.
void	**setMargin**(Insets m) Sets the margin between the menubar's border and its menus.
void	**setSelected**(Component sel) Sets the currently selected component, producing a a change to the selection model.

Class JMenuItem

An implementation of an item in a menu. A menu item is essentially a button sitting in a list. When the user selects the "button", the action associated with the menu item is performed. A JMenuItem contained in a JPopupMenu performs exactly that function.

Constructor Summary
JMenuItem() Creates a JMenuItem with no set text or icon.
JMenuItem(Icon icon) Creates a JMenuItem with the specified icon.
JMenuItem(String text) Creates a JMenuItem with the specified text.
JMenuItem(String text, Icon icon) Creates a JMenuItem with the specified text and icon.

JMenuItem(String text, int mnemonic) Creates a JMenuItem with the specified text and keyboard mnemonic.

Method Summary	
Component	**getComponent**() Returns the java.awt.Component used to paint this object.
MenuElement[]	**getSubElements**() This method returns an array containing the sub-menu components for this menu component.
protected void	**init**(String text, Icon icon) Initializes the menu item with the specified text and icon.
void	**menuSelectionChanged**(boolean isIncluded) Called by the MenuSelectionManager when the MenuElement is selected or unselected.

Class JScrollBar

An implementation of a scrollbar. The user positions the knob in the scrollbar to determine the contents of the viewing area. The program typically adjusts the display so that the end of the scrollbar represents the end of the displayable contents, or 100% of the contents. The start of the scrollbar is the beginning of the displayable contents, or 0%. The position of the knob within those bounds then translates to the corresponding percentage of the displayable contents.

Constructor Summary
JScrollBar() Creates a vertical scrollbar with the following initial values:
JScrollBar(int orientation) Creates a scrollbar with the specified orientation and the following initial values:
JScrollBar(int orientation, int value, int extent, int min, int max) Creates a scrollbar with the specified orientation, value, extent, minimum, and maximum.

Method Summary	
int	**getBlockIncrement**() For backwards compatibility with java.awt.Scrollbar.
int	**getBlockIncrement**(int direction) Returns the amount to change the scrollbar's value by, given a block (usually "page") up/down request.
int	**getMaximum**() The maximum value of the scrollbar is maximum - extent.
Dimension	**getMaximumSize**() The scrollbar is flexible along it's scrolling axis and rigid along the other

	axis.
int	**getMinimum**() Returns the minimum value supported by the scrollbar (usually zero).
Dimension	**getMinimumSize**() The scrollbar is flexible along it's scrolling axis and rigid along the other axis.
int	**getOrientation**() Returns the component's orientation (horizontal or vertical).
int	**getUnitIncrement**() For backwards compatibility with java.awt.Scrollbar.
int	**getUnitIncrement**(int direction) Returns the amount to change the scrollbar's value by, given a unit up/down request.
int	**getValue**() Returns the scrollbar's value.
boolean	**getValueIsAdjusting**() True if the scrollbar knob is being dragged.
void	**setBlockIncrement**(int blockIncrement) Sets the blockIncrement property.
void	**setEnabled**(boolean x) Enables the component so that the knob position can be changed.
void	**setMaximum**(int maximum) Sets the model's maximum property.
void	**setMinimum**(int minimum) Sets the model's minimum property.
void	**setOrientation**(int orientation) Set the scrollbar's orientation to either VERTICAL or HORIZONTAL.
void	**setUnitIncrement**(int unitIncrement) Sets the unitIncrement property.
void	**setValue**(int value) Sets the scrollbar's value.
void	**setValueIsAdjusting**(boolean b) Sets the model's valueIsAdjusting property.
void	**setValues**(int newValue, int newExtent, int newMin, int newMax) Sets the four BoundedRangeModel properties after forcing the arguments to obey the usual constraints:

Event Handling

The Event Model

The model for event processing in version 1.0 of the AWT is based on inheritance. In order for a program to catch and process GUI events, it must subclass GUI components and override either action() or

handleEvent() methods. Returning "true" from one of these methods consumes the event so it is not processed further; otherwise the event is propagated sequentially up the GUI hierarchy until either it is consumed or the root of the hierarchy is reached. The result of this model is that programs have essentially two choices for structuring their event-handling code:

1. Each individual component can be subclassed to specifically handle its target events. The result of this is a plethora of classes.
2. All events for an entire hierarchy (or subset thereof) can be handled by a particular container; the result is that the container's overridden action() or handleEvent() method must contain a complex conditional statement in order to process the events.

Issues with the Event Model

While the above model works fine for small applets with simple interfaces, it does not scale well for larger java programs for the following reasons:

- The requirement to subclass a component in order make any real use of its functionality is cumbersome to developers; subclassing should be reserved for circumstances where components are being extended in some functional or visual way.
- The inheritance model does not lend itself well to maintaining a clean separation between the application model and the GUI because application code must be integrated directly into the subclassed components at some level.
- Since ALL event types are filtered through the same methods, the logic to process the different event types (and possibly the event targets in approach #2) is complex and error-prone. It is not uncommon for programs to have perplexing bugs that are the result of returning an incorrect result (true or false) from the handleEvent() method. This becomes an even greater problem as new event types are added to the AWT; if the logic of existing handleEvent() methods isn't set up to deal properly with unknown types, programs could potentially break in very unpredictable ways.
- There is no filtering of events. Events are always delivered to components regardless of whether the components actually handle them or not. This is a general performance problem, particularly with high-frequency type events such as mouse moves.
- For many components, the action() method passes a String parameter which is equivalent to either the label of the component (Button, MenuItem) or the item selected (List, Choice). For programs which use approach #2, this often leads to poor coding and unwieldy string-compare logic that doesn't localize well.

The Delegation Model

Version 1.1 of the Java™ platform introduced a new delegation-based event model in AWT in order to:

- Resolve the problems mentioned previously
- Provide a more robust framework to support more complex java programs.

Design Goals

The primary design goals of the new model in the AWT are the following:

- Simple and easy to learn
- Support a clean separation between application and GUI code
- Facilitate the creation of robust event handling code which is less error-prone (strong compile-time checking)
- Flexible enough to enable varied application models for event flow and propagation
- For visual tool builders, enable run-time discovery of both events that a component generates as well as the events it may observe
- Support backward binary compatibility with the old model

Note: These goals are described from the particular perspective of the AWT. Since this model has also been designed to accommodate the JavaBeans architecture, the design goals from the JavaBeans perspective are described in the "Events" section of the JavaBeans Specification and may vary slightly from these goals.

Delegation Model Overview

Event types are encapsulated in a class hierarchy rooted at java.util.EventObject. An event is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated.

A Listener is an object that implements a specific EventListener interface extended from the generic java.util.EventListener. An EventListener interface defines one or more methods which are to be invoked by the event source in response to each specific event type handled by the interface.

An Event Source is an object which originates or "fires" events. The source defines the set of events it emits by providing a set of set<EventType>Listener (for single-cast) and/or add<EventType>Listener (for mult-cast) methods which are used to register specific listeners for those events.

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

Events :-

In an delegation model, an event is an object that describes a state change in a source. Some of the activities that causes events are pressing a button, entering a character, selecting item and clicking the mouse.

Low-level vs. Semantic Events

The AWT provides two conceptual types of events: low-level and semantic.

A low-level event is one which represents a low-level input or window-system occurrence on a visual component on the screen. The low-level event classes defined by the AWT are as follows:

java.util.EventObject
- java.awt.AWTEvent
  - java.awt.event.ComponentEvent (component resized, moved, etc.)
    - java.awt.event.FocusEvent (component got focus, lost focus)
    - java.awt.event.InputEvent
      - java.awt.event.KeyEvent (component got key-press, key-release, etc.)
      - java.awt.event.MouseEvent (component got mouse-down, mouse-move, etc.)
    - java.awt.event.ContainerEvent
    - java.awt.event.WindowEvent

Semantic events are defined at a higher-level to encapsulate the semantics of a user interface component's model. The semantic event classes defined by the AWT are as follows:

java.util.EventObject
- java.awt.AWTEvent
  - java.awt.event.ActionEvent ("do a command")
  - java.awt.event.AdjustmentEvent ("value was adjusted")
  - java.awt.event.ItemEvent ("item state has changed")
  - java.awt.event.TextEvent ("the value of the text object changed")

Note that these semantic events are not tied to specific screen-based component classes, but may apply across a set of components which implement a similar semantic model. For example, a Button object will fire an "action" event when it is pressed, a List object will fire an "action" event when an item is double-clicked, a MenuItem will fire an "action" event when it was selected from a menu, and a non-visual Timer object might fire an "action" when its timer goes off (the latter is a hypothetical case).

Event Listeners

An EventListener interface will typically have a separate method for each distinct event type the event class represents. So in essence, particular event semantics are defined by the combination of an Event class

paired with a particular method in an EventListener. For example, the FocusListener interface defines two methods, focusGained() and focusLost(), one for each event type that FocusEvent class represents.
The API attempts to define a balance between providing a reasonable granularity of Listener interface types and not providing a separate interface for every single event type.
The low-level listener interfaces defined by the AWT are as follows:

java.util.EventListener
- java.awt.event.ComponentListener
- java.awt.event.ContainerListener
- java.awt.event.FocusListener
- java.awt.event.KeyListener
- java.awt.event.MouseListener
- java.awt.event.MouseMotionListener
- java.awt.event.WindowListener

The semantic listener interfaces defined by the AWT are as follows:

java.util.EventListener
- java.awt.event.ActionListener
- java.awt.event.AdjustmentListener
- java.awt.event.ItemListener
- java.awt.event.TextListener

Event Sources

Because the events fired by an event source are defined by particular methods on that object, it is completely clear from the API documentation (as well as by using run-time introspection techniques) exactly which events an object supports.
All AWT event sources support a multicast model for listeners. This means that multiple listeners can be added and removed from a single source. **The API makes no guarantees about the order in which the events are delivered to a set of registered listeners for a given event on a given source.** Additionally, any event which allows its properties to be modified (via setXXX() methods) will be explicitly copied such that each listener receives a replica of the original event. If the order in which events are delivered to listeners is a factor for your program, you should chain the listeners off a single listener which is registered on the source (the fact that the event data is encapsulated in a single object makes propagating the event extremely simple).
Event delivery is synchronous (as with 1.0's handleEvent()), however programs should not make the assumption that the delivery of an event to a set of listeners will occur on the same thread.
Once again, a distinction is drawn between low-level and semantic events. For low-level events, the source will be one of the visual component classes (Button, Scrollbar, etc) since the event is tightly bound to the actual component on the screen. The low-level listeners are defined on the following components:

- java.awt.Component
  - addComponentListener(ComponentListener l)
  - addFocusListener(FocusListener l)
  - addKeyListener(KeyListener l)
  - addMouseListener(MouseListener l)
  - addMouseMotionListener(MouseMotionListener l)
- java.awt.Container
  - addContainerListener(ContainerListener l)
- java.awt.Dialog
  - addWindowListener(WindowListener l)
- java.awt.Frame
  - addWindowListener(WindowListener l)

For semantic events, the source is typically a higher-level interface representing the semantic model (and this higher-level interface is commonly `implemented' by components using the model). Following are the semantic listeners defined for AWT components:

- java.awt.Button
  addActionListener(ActionListener l)
- java.awt.Choice (implements java.awt.ItemSelectable)
  addItemListener(ItemListener l)
- java.awt.Checkbox (implements java.awt.ItemSelectable)
  addItemListener(ItemListener l)
- java.awt.CheckboxMenuItem (implements java.awt.ItemSelectable)
  addItemListener(ItemListener l)
- java.awt.List (implements java.awt.ItemSelectable)
  addActionListener(ActionListener l)
  addItemListener(ItemListener l)
- java.awt.MenuItem
  addActionListener(ActionListener l)
- java.awt.Scrollbar (implements java.awt.Adjustable)
  addAdjustmentListener(AdjustmentListener l)
- java.awt.TextArea
  addTextListener(TextListener l)
- java.awt.TextField
  addActionListener(ActionListener l)
  addTextListener(TextListener l)

EVENT CLASSES

1] Class ActionEvent

java.lang.Object
└java.util.EventObject
└java.awt.AWTEvent
└**java.awt.event.ActionEvent**

A semantic event which indicates that a component-defined action occured. This high-level event is generated by a component (such as a Button) when the component-specific action occurs (such as being pressed). The event is passed to every every ActionListener object that registered to receive such events using the component's addActionListener method.
The object that implements the ActionListener interface gets this ActionEvent when the event occurs. The listener is therefore spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "button pressed".

Field	
static int	**ACTION_PERFORMED** This event id indicates that a meaningful action occured.
static int	**ALT_MASK** The alt modifier.
static int	**CTRL_MASK** The control modifier.

static int	**SHIFT_MASK** The shift modifier.

Constructor
ActionEvent(Object source, int id, String command) Constructs an ActionEvent object.
ActionEvent(Object source, int id, String command, int modifiers) Constructs an ActionEvent object with modifier keys.
ActionEvent(Object source, int id, String command, long when, int modifiers) Constructs an ActionEvent object with the specified modifier keys and timestamp.

Method	
String	**getActionCommand**() Returns the command string associated with this action.
int	**getModifiers**() Returns the modifier keys held down during this action event.
long	**getWhen**() Returns the timestamp of when this event occurred.

2] Class AdjustmentEvent

java.lang.Object
└ java.util.EventObject
└ java.awt.AWTEvent
└ **java.awt.event.AdjustmentEvent**

This event is generated by a scrollbar. This event can be generated by five ways. It defines five integer constants which are listed below:

Field	
static int	**BLOCK_DECREMENT** It is set when you click inside the scrollbar to decrease its value. Its value is 3.
static int	**BLOCK_INCREMENT** It is set when you click inside the scrollbar to increase its value. Its value is 4.
static int	**TRACK** It is set when the slider of the scrollbar is dragged to adjust value. Its value is 5.
static int	**UNIT_DECREMENT** It is set when you click the button at the end of the scrollbar to increase its value. Its value is 1.
static int	**UNIT_INCREMENT** It is set when you click the button at the end of the scrollbar to decrease its value. Its value is 2

It has another integer constant which ADJUSTMENT_VALUE_CHANGED which indicates that the value is adjusted. The constructor is:

Constructor
AdjustmentEvent(Adjustable source, int id, int type, int value) Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type, and value.

Method	
Adjustable	**getAdjustable**() Returns the Adjustable object where this event originated.
int	**getAdjustmentType**() Returns the type of adjustment which caused the value changed event.
int	**getValue**() Returns the current value in the adjustment event.

3] Class ComponentEvent

java.lang.Object
└ java.util.EventObject
└ java.awt.AWTEvent
└ **java.awt.event.ComponentEvent**

A low-level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component-level events).
Component events are provided for notification purposes ONLY; The AWT will automatically handle component moves and resizes internally so that GUI layout works properly regardless of whether a program is receiving these events or not.
In addition to serving as the base class for other component-related events (InputEvent, FocusEvent, WindowEvent, ContainerEvent), this class defines the events that indicate changes in a component's size, position, or visibility.
This low-level event is generated by a component object (such as a List) when the component is moved, resized, rendered invisible, or made visible again. The event is passed to every ComponentListener or ComponentAdapter object which registered to receive such events using the component's addComponentListener method. (ComponentAdapter objects implement the ComponentListener interface.) Each such listener object gets this ComponentEvent when the event occurs.

Field	
static int	**COMPONENT_HIDDEN** This event indicates that the component was rendered invisible.
static int	**COMPONENT_MOVED** This event indicates that the component's position changed.
static int	**COMPONENT_RESIZED** This event indicates that the component's size changed.
static int	**COMPONENT_SHOWN** This event indicates that the component was made visible.

Constructor
ComponentEvent(Component source, int id) Constructs a ComponentEvent object.

Method	
Component	**getComponent**() Returns the originator of the event.

4] Class ContainerEvent

java.lang.Object
└ java.util.EventObject
└ java.awt.AWTEvent
└ java.awt.event.ComponentEvent
└ **java.awt.event.ContainerEvent**

A low-level event which indicates that a container's contents changed because a component was added or removed.
Container events are provided for notification purposes ONLY; The AWT will automatically handle changes to the containers contents internally so that the program works properly regardless of whether the program is receiving these events or not.
This low-level event is generated by a container object (such as a Panel) when a component is added to it or removed from it. The event is passed to every ContainerListener or ContainerAdapter object which registered to receive such events using the component's addContainerListener method. (ContainerAdapter objects implement the ContainerListener interface.) Each such listener object gets this ContainerEvent when the event occurs.

Field	
static int	**COMPONENT_ADDED** This event indicates that a component was added to the container.
static int	**COMPONENT_REMOVED** This event indicates that a component was removed from the container.

Constructor
ContainerEvent(Component source, int id, Component child) Constructs a ContainerEvent object.

Method	
Component	**getChild**() Returns the component that was affected by the event.
Container	**getContainer**() Returns the originator of the event.

5] Class FocusEvent

A low-level event which indicates that a Component has gained or lost the input focus. This low-level event is generated by a Component (such as a TextField). The event is passed to every FocusListener or FocusAdapter object which registered to receive such events using the Component's addFocusListener method. (FocusAdapter objects implement the FocusListener interface.) Each such listener object gets this FocusEvent when the event occurs.
There are two levels of focus events: permanent and temporary. Permanent focus change events occur when focus is directly moved from one Component to another, such as through a call to requestFocus() or

as the user uses the TAB key to traverse Components. Temporary focus change events occur when focus is temporarily lost for a Component as the indirect result of another operation, such as Window deactivation or a Scrollbar drag. In this case, the original focus state will automatically be restored once that operation is finished, or, for the case of Window deactivation, when the Window is reactivated. Both permanent and temporary focus events are delivered using the FOCUS_GAINED and FOCUS_LOST event ids; the level may be distinguished in the event using the isTemporary() method.

Field	
static int	**FOCUS_GAINED** This event indicates that the Component is now the focus owner.
static int	**FOCUS_LOST** This event indicates that the Component is no longer the focus owner.

Constructor
FocusEvent(Component source, int id) Constructs a FocusEvent object and identifies it as a permanent change in focus.
FocusEvent(Component source, int id, boolean temporary) Constructs a FocusEvent object and identifies whether or not the change is temporary.
FocusEvent(Component source, int id, boolean temporary, Component opposite) Constructs a FocusEvent object with the specified temporary state and opposite Component.

Method	
Component	**getOppositeComponent**() Returns the other Component involved in this focus change.
boolean	**isTemporary**() Identifies the focus change event as temporary or permanent.

6] Class InputEvent

The root event class for all component-level input events. Input events are delivered to listeners before they are processed normally by the source where they originated. This allows listeners and component subclasses to "consume" the event so that the source will not process them in their default manner. For example, consuming mousePressed events on a Button component will prevent the Button from being activated.

Field	
static int	**ALT_DOWN_MASK** The Alt key extended modifier constant.
static int	**ALT_GRAPH_DOWN_MASK** The AltGraph key extended modifier constant.
static int	**ALT_GRAPH_MASK** The AltGraph key modifier constant.
static int	**ALT_MASK** The Alt key modifier constant.

static int	**BUTTON1_DOWN_MASK** The Mouse Button1 extended modifier constant.
static int	**BUTTON1_MASK** The Mouse Button1 modifier constant.
static int	**BUTTON2_DOWN_MASK** The Mouse Button2 extended modifier constant.
static int	**BUTTON2_MASK** The Mouse Button2 modifier constant.
static int	**BUTTON3_DOWN_MASK** The Mouse Button3 extended modifier constant.
static int	**BUTTON3_MASK** The Mouse Button3 modifier constant.
static int	**CTRL_DOWN_MASK** The Control key extended modifier constant.
static int	**CTRL_MASK** The Control key modifier constant.
static int	**META_DOWN_MASK** The Meta key extended modifier constant.
static int	**META_MASK** The Meta key modifier constant.
static int	**SHIFT_DOWN_MASK** The Shift key extended modifier constant.
static int	**SHIFT_MASK** The Shift key modifier constant.
Method	
boolean	**isAltDown**() Used to test if the alter was pressed at the time an event is generated or not.
boolean	**isAltGraphDown**() Used to test if the alterGraph was pressed at the time an event is generated or not..
boolean	**isControlDown**() Used to test if the control was pressed at the time an event is generated or not.
boolean	**isMetaDown**() Used to test if the meta was pressed at the time an event is generated or not.
boolean	**isShiftDown**() Used to test if the shift was pressed at the time an event is generated or not.
int	**getModifiers()** **It obtains a value that contains all of the original modifier flags.**
int	**getModifiersEX()** **It obtains extended modifiers flags.**

7] Class KeyEvent

An event which indicates that a keystroke occurred in a component.
This low-level event is generated by a component object (such as a text field) when a key is pressed, released, or typed. The event is passed to every KeyListener or KeyAdapter object which registered to receive such events using the component's addKeyListener method. (KeyAdapter objects implement the KeyListener interface.) Each such listener object gets this KeyEvent when the event occurs.
"Key typed" events are higher-level and generally do not depend on the platform or keyboard layout. They are generated when a Unicode character is entered, and are the preferred way to find out about character input. In the simplest case, a key typed event is produced by a single key press (e.g., 'a'). Often, however, characters are produced by series of key presses (e.g., 'shift' + 'a'), and the mapping from key pressed events to key typed events may be many-to-one or many-to-many. Key releases are not usually necessary to generate a key typed event, but there are some cases where the key typed event is not generated until a key is released (e.g., entering ASCII sequences via the Alt-Numpad method in Windows). No key typed events are generated for keys that don't generate Unicode characters (e.g., action keys, modifier keys, etc.). The getKeyChar method always returns a valid Unicode character or CHAR_UNDEFINED. For key pressed and key released events, the getKeyCode method returns the event's keyCode. For key typed events, the getKeyCode method always returns VK_UNDEFINED.
"Key pressed" and "key released" events are lower-level and depend on the platform and keyboard layout. They are generated whenever a key is pressed or released, and are the only way to find out about keys that don't generate character input (e.g., action keys, modifier keys, etc.). The key being pressed or released is indicated by the getKeyCode method, which returns a virtual key code.
Virtual key codes are used to report which keyboard key has been pressed, rather than a character generated by the combination of one or more keystrokes (such as "A", which comes from shift and "a"). For example, pressing the Shift key will cause a KEY_PRESSED event with a VK_SHIFT keyCode, while pressing the 'a' key will result in a VK_A keyCode. After the 'a' key is released, a KEY_RELEASED event will be fired with VK_A. Separately, a KEY_TYPED event with a keyChar value of 'A' is generated.
Notes:

- Key combinations which do not result in Unicode characters, such as action keys like F1 and the HELP key, do not generate KEY_TYPED events.
- Not all keyboards or systems are capable of generating all virtual key codes. No attempt is made in Java to generate these keys artificially.
- Virtual key codes do not identify a physical key: they depend on the platform and keyboard layout. For example, the key that generates VK_Q when using a U.S. keyboard layout will generate VK_A when using a French keyboard layout.
- Not all characters have a keycode associated with them. For example, there is no keycode for the question mark because there is no keyboard for which it appears on the primary layer.
- In order to support the platform-independent handling of action keys, the Java platform uses a few additional virtual key constants for functions that would otherwise have to be recognized by interpreting virtual key codes and modifiers. For example, for Japanese Windows keyboards, VK_ALL_CANDIDATES is returned instead of VK_CONVERT with the ALT modifier.

WARNING: Aside from those keys that are defined by the Java language (VK_ENTER, VK_BACK_SPACE, and VK_TAB), do not rely on the values of the VK_ constants. Sun reserves the right to change these values as needed to accomodate a wider range of keyboards in the future.

Field	
static int	**KEY_PRESSED** The "key pressed" event.

static int	**KEY_RELEASED** The "key released" event.
static int	**KEY_TYPED** The "key typed" event.
static int	**VK_0** to **VK_9** VK_0 thru VK_9 are the same as ASCII '0' thru '9' (0x30 - 0x39)
static int	**VK_A** to **VK_Z** VK_A thru VK_Z are the same as ASCII 'A' thru 'Z' (0x41 - 0x5A)
static int	**VK_ALT**
static int	**VK_CANCEL**
static int	**VK_DOWN** Constant for the non-numpad **down** arrow key.
static int	**VK_F1** to **VK_F12** Constant for the F1 to F12 function key.
static int	**VK_LEFT** Constant for the non-numpad **left** arrow key.
static int	**VK_PAGE_DOWN**
static int	**VK_PAGE_UP**
static int	**VK_RIGHT** Constant for the non-numpad **right** arrow key.
static int	**VK_SHIFT**
static int	**VK_UP** Constant for the non-numpad **up** arrow key.

Constructor
KeyEvent(Component source, int id, long when, int modifiers, int keyCode)
KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar) Constructs a KeyEvent object.

Method	
char	**getKeyChar**() Returns the character associated with the key in this event.
int	**getKeyCode**() Returns the integer keyCode associated with the key in this event.

int	**getKeyLocation**() Returns the location of the key that originated this key event.

8] Class MouseEvent

An event which indicates that a mouse action occurred in a component. A mouse action is considered to occur in a particular component if and only if the mouse cursor is over the unobscured part of the component's bounds when the action happens. Component bounds can be obscurred by the visible component's children or by a menu or by a top-level window. This event is used both for mouse events (click, enter, exit) and mouse motion events (moves and drags).

This low-level event is generated by a component object for:

- Mouse Events
  - a mouse button is pressed
  - a mouse button is released
  - a mouse button is clicked (pressed and released)
  - the mouse cursor enters the unobscured part of component's geometry
  - the mouse cursor exits the unobscured part of component's geometry
- Mouse Motion Events
  - the mouse is moved
  - the mouse is dragged

A MouseEvent object is passed to every MouseListener or MouseAdapter object which is registered to receive the "interesting" mouse events using the component's addMouseListener method. (MouseAdapter objects implement the MouseListener interface.) Each such listener object gets a MouseEvent containing the mouse event.

A MouseEvent object is also passed to every MouseMotionListener or MouseMotionAdapter object which is registered to receive mouse motion events using the component's addMouseMotionListener method. (MouseMotionAdapter objects implement the MouseMotionListener interface.) Each such listener object gets a MouseEvent containing the mouse motion event.

When a mouse button is clicked, events are generated and sent to the registered MouseListeners. The state of modal keys can be retrieved using InputEvent.getModifiers() and InputEvent.getModifiersEx(). The button mask returned by InputEvent.getModifiers() reflects only the button that changed state, not the current state of all buttons. (Note: Due to overlap in the values of ALT_MASK/BUTTON2_MASK and META_MASK/BUTTON3_MASK, this is not always true for mouse events involving modifier keys). To get the state of all buttons and modifier keys, use InputEvent.getModifiersEx(). The button which has changed state is returned by getButton()

For example, if the first mouse button is pressed, events are sent in the following order:

ID	Modifiers	Button
MOUSE_PRESSED:	BUTTON1_MASK	BUTTON1
MOUSE_RELEASED:	BUTTON1_MASK	BUTTON1
MOUSE_CLICKED:	BUTTON1_MASK	BUTTON1

When multiple mouse buttons are pressed, each press, release, and click results in a separate event.

For example, if the user presses **button 1** followed by **button 2**, and then releases them in the same order, the following sequence of events is generated:

ID	Modifiers	Button
MOUSE_PRESSED:	BUTTON1_MASK	BUTTON1
MOUSE_PRESSED:	BUTTON2_MASK	BUTTON2
MOUSE_RELEASED:	BUTTON1_MASK	BUTTON1
MOUSE_CLICKED:	BUTTON1_MASK	BUTTON1
MOUSE_RELEASED:	BUTTON2_MASK	BUTTON2

MOUSE_CLICKED: BUTTON2_MASK BUTTON2

If **button 2** is released first, the MOUSE_RELEASED/MOUSE_CLICKED pair for BUTTON2_MASK arrives first, followed by the pair for BUTTON1_MASK.

MOUSE_DRAGGED events are delivered to the Component in which the mouse button was pressed until the mouse button is released (regardless of whether the mouse position is within the bounds of the Component). Due to platform-dependent Drag&Drop implementations, MOUSE_DRAGGED events may not be delivered during a native Drag&Drop operation. In a multi-screen environment mouse drag events are delivered to the Component even if the mouse position is outside the bounds of the GraphicsConfiguration associated with that Component. However, the reported position for mouse drag events in this case may differ from the actual mouse position:

- In a multi-screen environment without a virtual device:
  The reported coordinates for mouse drag events are clipped to fit within the bounds of the GraphicsConfiguration associated with the Component.
- In a multi-screen environment with a virtual device:
  The reported coordinates for mouse drag events are clipped to fit within the bounds of the virtual device associated with the Component.

Field	
static int	**MOUSE_CLICKED** The "mouse clicked" event.
static int	**MOUSE_DRAGGED** The "mouse dragged" event.
static int	**MOUSE_ENTERED** The "mouse entered" event.
static int	**MOUSE_EXITED** The "mouse exited" event.
static int	**MOUSE_MOVED** The "mouse moved" event.
static int	**MOUSE_PRESSED** The "mouse pressed" event.
static int	**MOUSE_RELEASED** The "mouse released" event.
static int	**MOUSE_WHEEL** The "mouse wheel" event.

Constructor
MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger) Constructs a MouseEvent object with the specified source component, type, modifiers, coordinates, and click count.

Method	
int	**getButton**()

	Returns which, if any, of the mouse buttons has changed state.
int	**getClickCount**() Returns the number of mouse clicks associated with this event.
static String	**getMouseModifiersText**(int modifiers) Returns a String describing the modifier keys and mouse buttons that were down during the event, such as "Shift", or "Ctrl+Shift".
Point	**getPoint**() Returns the x,y position of the event relative to the source component.
int	**getX**() Returns the horizontal x position of the event relative to the source component.
int	**getY**() Returns the vertical y position of the event relative to the source component.
boolean	**isPopupTrigger**() Returns whether or not this mouse event is the popup menu trigger event for the platform.

9] Class MouseWheelEvent

An event which indicates that the mouse wheel was rotated in a component.
A wheel mouse is a mouse which has a wheel in place of the middle button. This wheel can be rotated towards or away from the user. Mouse wheels are most often used for scrolling, though other uses are possible.
A MouseWheelEvent object is passed to every MouseWheelListener object which registered to receive the "interesting" mouse events using the component's addMouseWheelListener method. Each such listener object gets a MouseEvent containing the mouse event.
Due to the mouse wheel's special relationship to scrolling Components, MouseWheelEvents are delivered somewhat differently than other MouseEvents. This is because while other MouseEvents usually affect a change on the Component directly under the mouse cursor (for instance, when clicking a button), MouseWheelEvents often have an effect away from the mouse cursor (moving the wheel while over a Component inside a ScrollPane should scroll one of the Scrollbars on the ScrollPane).
MouseWheelEvents start delivery from the Component underneath the mouse cursor. If MouseWheelEvents are not enabled on the Component, the event is delivered to the first ancestor Container with MouseWheelEvents enabled. This will usually be a ScrollPane with wheel scrolling enabled. The source Component and x,y coordinates will be relative to the event's final destination (the ScrollPane). This allows a complex GUI to be installed without modification into a ScrollPane, and for all MouseWheelEvents to be delivered to the ScrollPane for scrolling. Some AWT Components are implemented using native widgets which display their own scrollbars and handle their own scrolling. The particular Components for which this is true will vary from platform to platform. When the mouse wheel is moved over one of these Components, the event is delivered straight to the native widget, and not propagated to ancestors. Platforms offer customization of the amount of scrolling that should take place when the mouse wheel is moved. The two most common settings are to scroll a certain number of "units" (commonly lines of text in a text-based component) or an entire "block" (similar to page-up/page-down). The MouseWheelEvent offers methods for conforming to the underlying platform settings. These platform settings can be changed at any time by the user. MouseWheelEvents reflect the most recent settings.

Field	
static int	**WHEEL_BLOCK_SCROLL**

	Constant representing scrolling by a "block" (like scrolling with page-up, page-down keys)
static int	**WHEEL_UNIT_SCROLL** Constant representing scrolling by "units" (like scrolling with the arrow keys)

Constructor
MouseWheelEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger, int scrollType, int scrollAmount, int wheelRotation) Constructs a MouseWheelEvent object with the specified source component, type, modifiers, coordinates, scroll type, scroll amount, and wheel rotation.

Method	
int	**getScrollAmount**() Returns the number of units that should be scrolled in response to this event.
int	**getScrollType**() Returns the type of scrolling that should take place in response to this event.
int	**getWheelRotation**() Returns the number of "clicks" the mouse wheel was rotated.

10] Class WindowEvent

A low-level event that indicates that a window has changed its status. This low-level event is generated by a Window object when it is opened, closed, activated, deactivated, iconified, or deiconified, or when focus is transfered into or out of the Window.

The event is passed to every WindowListener or WindowAdapter object which registered to receive such events using the window's addWindowListener method. (WindowAdapter objects implement the WindowListener interface.) Each such listener object gets this WindowEvent when the event occurs.

Field Summary	
static int	**WINDOW_ACTIVATED** The window-activated event type.
static int	**WINDOW_CLOSED** The window closed event.
static int	**WINDOW_CLOSING** The "window is closing" event.
static int	**WINDOW_DEACTIVATED** The window-deactivated event type.
static int	**WINDOW_DEICONIFIED** The window deiconified event type.
static int	**WINDOW_GAINED_FOCUS** The window-gained-focus event type.
static int	**WINDOW_ICONIFIED** The window iconified event.

static int	**WINDOW_LOST_FOCUS** The window-lost-focus event type.
static int	**WINDOW_OPENED** The window opened event.
static int	**WINDOW_STATE_CHANGED** The window-state-changed event type.

Constructor
WindowEvent(Window source, int id) Constructs a WindowEvent object.
WindowEvent(Window source, int id, int oldState, int newState) Constructs a WindowEvent object with the specified previous and new window states.
WindowEvent(Window source, int id, Window opposite) Constructs a WindowEvent object with the specified opposite Window.
WindowEvent(Window source, int id, Window opposite, int oldState, int newState) Constructs a WindowEvent object.

Method	
int	**getNewState**() For WINDOW_STATE_CHANGED events returns the new state of the window.
int	**getOldState**() For WINDOW_STATE_CHANGED events returns the previous state of the window.
Window	**getOppositeWindow**() Returns the other Window involved in this focus or activation change.
Window	**getWindow**() Returns the originator of the event.

11] Class ItemEvent

A semantic event which indicates that an item was selected or deselected. This high-level event is generated by an ItemSelectable object (such as a List) when an item is selected or deselected by the user. The event is passed to every ItemListener object which registered to receive such events using the component's addItemListener method.

The object that implements the ItemListener interface gets this ItemEvent when the event occurs. The listener is spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "item selected" or "item deselected".

Field	
static int	**DESELECTED** This state-change-value indicates that a selected item was deselected.
static int	**ITEM_STATE_CHANGED** This event id indicates that an item's state changed.
static int	**SELECTED**

	This state-change value indicates that an item was selected.

Constructor
ItemEvent(ItemSelectable source, int id, Object item, int stateChange) Constructs an ItemEvent object.

Method	
Object	**getItem**() Returns the item affected by the event.
ItemSelectable	**getItemSelectable**() Returns the originator of the event.
int	**getStateChange**() Returns the type of state change (selected or deselected).

12] Class TextEvent

A semantic event which indicates that an object's text changed. This high-level event is generated by an object (such as a TextComponent) when its text changes. The event is passed to every TextListener object which registered to receive such events using the component's addTextListener method.
The object that implements the TextListener interface gets this TextEvent when the event occurs. The listener is spared the details of processing individual mouse movements and key strokes Instead, it can process a "meaningful" (semantic) event like "text changed".

Field	
static int	**TEXT_FIRST** The first number in the range of ids used for text events.
static int	**TEXT_LAST** The last number in the range of ids used for text events.
static int	**TEXT_VALUE_CHANGED** This event id indicates that object's text changed.

Constructor
TextEvent(Object source, int id) Constructs a TextEvent object.

Method	
String	**paramString**() Returns a parameter string identifying this text event.

The sources of events

- **Button**

It generates the action events (ActionEvent class) when the button is pressed.

- **Checkbox**

It generates item events (ItemEvent class) when the check box is checked or unchecked.

- **Choice**

It generates item events (ItemEvent class) when the item is changed in the choice.

- **List**

It can generate action events as well as item events. When an item is double-clicked, it generates ActionEvent and it generates ItemEvent when an item is selected or deselected in the list.

- **Menu Item**

It can generate action events as well as item events. It generates action events when a menu item is selected from a menu and generates an item event when a checkable menu item is selected or deselected.

- **Scrollbar**

It generates adjustment events (AdjustmentEvent class) when the scroll bar is adjusted.

- **Text controls such as text field, text area:**

It generates text event when the user enters a character into the text field or text area.

- **Window**

It generates window events when a window is activated, opened, closed etc.

EVENT LISTENERS

1] Interface ActionListener

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.

Method	
void	**actionPerformed**(ActionEvent e) Invoked when an action occurs.

2] Interface AdjustmentListener

The listener interface for receiving adjustment events.

Method	
void	**adjustmentValueChanged**(AdjustmentEvent e) Invoked when the value of the adjustable has changed.

3] Interface ComponentListener

The listener interface for receiving component events. The class that is interested in processing a component event either implements this interface (and all the methods it contains) or extends the abstract ComponentAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a component using the component's addComponentListener method. When the component's size, location, or visibility changes, the relevant method in the listener object is invoked, and the ComponentEvent is passed to it.
Component events are provided for notification purposes ONLY; The AWT will automatically handle component moves and resizes internally so that GUI layout works properly regardless of whether a program registers a ComponentListener or not.

Method	
void	**componentHidden**(ComponentEvent e) Invoked when the component has been made invisible.
void	**componentMoved**(ComponentEvent e) Invoked when the component's position changes.

void	**componentResized**(ComponentEvent e) Invoked when the component's size changes.
void	**componentShown**(ComponentEvent e) Invoked when the component has been made visible.

4] Interface ContainerListener

The listener interface for receiving container events. The class that is interested in processing a container event either implements this interface (and all the methods it contains) or extends the abstract ContainerAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a component using the component's addContainerListener method. When the container's contents change because a component has been added or removed, the relevant method in the listener object is invoked, and the ContainerEvent is passed to it.
Container events are provided for notification purposes ONLY; The AWT will automatically handle add and remove operations internally so the program works properly regardless of whether the program registers a ComponentListener or not.

Method	
void	**componentAdded**(ContainerEvent e) Invoked when a component has been added to the container.
void	**componentRemoved**(ContainerEvent e) Invoked when a component has been removed from the container.

5] Interface FocusListener

The listener interface for receiving keyboard focus events on a component. The class that is interested in processing a focus event either implements this interface (and all the methods it contains) or extends the abstract FocusAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a component using the component's addFocusListener method. When the component gains or loses the keyboard focus, the relevant method in the listener object is invoked, and the FocusEvent is passed to it.

Method	
void	**focusGained**(FocusEvent e) Invoked when a component gains the keyboard focus.
void	**focusLost**(FocusEvent e) Invoked when a component loses the keyboard focus.

6] Interface ItemListener

The listener interface for receiving item events. The class that is interested in processing an item event implements this interface. The object created with that class is then registered with a component using the component's addItemListener method. When an item-selection event occurs, the listener object's itemStateChanged method is invoked.

Method	
void	**itemStateChanged**(ItemEvent e) Invoked when an item has been selected or deselected by the user.

7] Interface KeyListener

The listener interface for receiving keyboard events (keystrokes). The class that is interested in processing a keyboard event either implements this interface (and all the methods it contains) or extends the abstract KeyAdapter class (overriding only the methods of interest).

The listener object created from that class is then registered with a component using the component's addKeyListener method. A keyboard event is generated when a key is pressed, released, or typed (pressedn and released). The relevant method in the listener object is then invoked, and the KeyEvent is passed to it.

Method	
void	**keyPressed**(KeyEvent e) Invoked when a key has been pressed.
void	**keyReleased**(KeyEvent e) Invoked when a key has been released.
void	**keyTyped**(KeyEvent e) Invoked when a key has been typed.

8] Interface MouseListener

The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component. (To track mouse moves and mouse drags, use the MouseMotionListener.)
The class that is interested in processing a mouse event either implements this interface (and all the methods it contains) or extends the abstract MouseAdapter class (overriding only the methods of interest).
The listener object created from that class is then registered with a component using the component's addMouseListener method. A mouse event is generated when the mouse is pressed, released clicked (pressed and released). A mouse event is also generated when the mouse cursor enters or leaves a component. When a mouse event occurs, the relevant method in the listener object is invoked, and the MouseEvent is passed to it.

Method	
void	**mouseClicked**(MouseEvent e) Invoked when the mouse button has been clicked (pressed and released) on a component.
void	**mouseEntered**(MouseEvent e) Invoked when the mouse enters a component.
void	**mouseExited**(MouseEvent e) Invoked when the mouse exits a component.
void	**mousePressed**(MouseEvent e) Invoked when a mouse button has been pressed on a component.
void	**mouseReleased**(MouseEvent e) Invoked when a mouse button has been released on a component.

9] Interface MouseMotionListener

The listener interface for receiving mouse motion events on a component. (For clicks and other mouse events, use the MouseListener.)
The class that is interested in processing a mouse motion event either implements this interface (and all the methods it contains) or extends the abstract MouseMotionAdapter class (overriding only the methods of interest).
The listener object created from that class is then registered with a component using the component's addMouseMotionListener method. A mouse motion event is generated when the mouse is moved or dragged. (Many such events will be generated). When a mouse motion event occurs, the relevant method in the listener object is invoked, and the MouseEvent is passed to it.

Method	
void	**mouseDragged**(MouseEvent e) Invoked when a mouse button is pressed on a component and then dragged.
void	**mouseMoved**(MouseEvent e) Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

10] Interface MouseWheelListener

The listener interface for receiving mouse wheel events on a component. (For clicks and other mouse events, use the MouseListener. For mouse movement and drags, use the MouseMotionListener.)
The class that is interested in processing a mouse wheel event implements this interface (and all the methods it contains).
The listener object created from that class is then registered with a component using the component's addMouseWheelListener method. A mouse wheel event is generated when the mouse wheel is rotated. When a mouse wheel event occurs, that object's mouseWheelMoved method is invoked.

Method	
void	**mouseWheelMoved**(MouseWheelEvent e) Invoked when the mouse wheel is rotated.

11] Interface TextListener

The listener interface for receiving text events. The class that is interested in processing a text event implements this interface. The object created with that class is then registered with a component using the component's addTextListener method. When the component's text changes, the listener object's textValueChanged method is invoked.

Method Summary	
void	**textValueChanged**(TextEvent e) Invoked when the value of the text has changed.

12] Interface WindowFocusListener

The listener interface for receiving WindowEvents, including WINDOW_GAINED_FOCUS and WINDOW_LOST_FOCUS events. The class that is interested in processing a WindowEvent either implements this interface (and all the methods it contains) or extends the abstract WindowAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a Window using the Window's addWindowFocusListener method. When the Window's status changes by virtue of it being opened, closed, activated, deactivated, iconified, or deiconified, or by focus being transfered into or out of the Window, the relevant method in the listener object is invoked, and the WindowEvent is passed to it.

Method	
void	**windowGainedFocus**(WindowEvent e) Invoked when the Window is set to be the focused Window, which means that the Window, or one of its subcomponents, will receive keyboard events.
void	**windowLostFocus**(WindowEvent e) Invoked when the Window is no longer the focused Window, which means that keyboard events will no longer be delivered to the Window or any of its subcomponents.

13] Interface WindowListener

The listener interface for receiving window events. The class that is interested in processing a window event either implements this interface (and all the methods it contains) or extends the abstract WindowAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a Window using the window's addWindowListener method. When the window's status changes by virtue of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the WindowEvent is passed to it.

Method	
void	**windowActivated**(WindowEvent e) Invoked when the Window is set to be the active Window.
void	**windowClosed**(WindowEvent e) Invoked when a window has been closed as the result of calling dispose on the window.
void	**windowClosing**(WindowEvent e) Invoked when the user attempts to close the window from the window's system menu.
void	**windowDeactivated**(WindowEvent e) Invoked when a Window is no longer the active Window.
void	**windowDeiconified**(WindowEvent e) Invoked when a window is changed from a minimized to a normal state.
void	**windowIconified**(WindowEvent e) Invoked when a window is changed from a normal to a minimized state.
void	**windowOpened**(WindowEvent e) Invoked the first time a window is made visible.

Adapter classes

The adapter classes are very special classes that are used to make event handling very easy.There are listener interfaces that have many methods for event handling. And as we know that by implementing an interface we have to implement all the methods of that interface. But sometimes we need only one or some methods of the interface. In that case,adapter classes are the best solution.

For example the MouseListener interface has five methods: mouseClicked(), mouseEntered(),mouseExited(),mousePressed(), and mousePressed(). If in your program, you just need two events:mouseEntered() and mouseExited(). But still you want to implement all of the five methods of this interface. So you can use the adapter class for the MouseListener interface.

The adapter classes contain an empty implementations for each method of the event listener interface. To use an adapter class, you have to extend that adapter class.java

Below table defines the adapter classes for the following listener interface Listener Interfaces	Adapter classes
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

The following example uses the adapter class for the MouseListener interface..

```
//java prg for using adapter class for MouseListener...
import java.applet.*;
```

```
import java.awt.*;
import java.awt.event.*;
/*
<applet code=adapterex width=300 height=150>
</applet>
*/
public class adapterex extends Applet
{
String msg="";
public void init()
{
//register mouse event
addMouseListener(new myadapter(this));
}
public void paint(Graphics g)
{
Font f=new Font("Verdana",Font.BOLD,24);
g.setFont(f);
g.setColor(Color.red);
g.drawString(msg,20,20);
}
}
class myadapter extends MouseAdapter
{
adapterex adapter; //create object of adapterex...
public myadapter(adapterex adapter)
{
this.adapter=adapter;
}
public void mouseEntered(MouseEvent obj)
{
adapter.msg="Mouse Entered.";//msg is member of adapterex...
adapter.setBackground(Color.blue);
adapter.repaint();//paint() method is in adapterex..
}
public void mouseExited(MouseEvent obj)
{
adapter.msg="Mouse Exited.";
adapter.setBackground(Color.green);
adapter.repaint();
}
}
```

Awt controls

AWT Label Class

Introduction

Label is a passive control because it does not create any event when accessed by the user. The label control is an object of Label. A label displays a single line of read-only text. However the text can be changed by the application programmer but cannot be changed by the end user in any way.

Class declaration

Following is the declaration for **java.awt.Label** class:

```
public class Label
  extends Component
    implements Accessible
```

Field

Following are the fields for **java.awt.Component** class:

- **static int CENTER** -- Indicates that the label should be centered.
- **static int LEFT** -- Indicates that the label should be left justified.
- **static int RIGHT** -- Indicates that the label should be right justified.

Class constructors

S.N.	Constructor & Description
1	**Label()** Constructs an empty label.
2	**Label(String text)** Constructs a new label with the specified string of text, left justified.
3	**Label(String text, int alignment)** Constructs a new label that presents the specified string of text with the specified alignment.

Class methods

S.N.	Method & Description
1	**void addNotify()** Creates the peer for this label.
2	**AccessibleContext getAccessibleContext()** Gets the AccessibleContext associated with this Label.
3	**int getAlignment()** Gets the current alignment of this label.
4	**String getText()** Gets the text of this label.
5	**protected String paramString()** Returns a string representing the state of this Label.
6	**void setAlignment(int alignment)** Sets the alignment for this label to the specified alignment.
7	**void setText(String text)** Sets the text for this label to the specified text.

AWT Button Class

Introduction

Button is a control component that has a label and generates an event when pressed. When a button is pressed and released, AWT sends an instance of ActionEvent to the button, by calling processEvent on the button. The button's processEvent method receives all events for the button; it passes an action event along by calling its own processActionEvent method. The latter method passes the action event on to any action listeners that have registered an interest in action events generated by this button.

If an application wants to perform some action based on a button being pressed and released, it should implement ActionListener and register the new listener to receive events from this button, by calling the button's addActionListener method. The application can make use of the button's action command as a messaging protocol.

Class declaration

Following is the declaration for **java.awt.Button** class:

```
public class Button
  extends Component
    implements Accessible
```

Class constructors

S.N.	Constructor & Description
1	**Button()** Constructs a button with an empty string for its label.

2	**Button(String text)** Constructs a new button with specified label.

AWT CheckBox Class

Introduction

Checkbox control is used to turn an option on(true) or off(false). There is label for each checkbox representing what the checkbox does.The state of a checkbox can be changed by clicking on it.

Class declaration

Following is the declaration for **java.awt.Checkbox** class:

```
public class Checkbox
  extends Component
    implements ItemSelectable,Accessible
```

Class constructors

S.N.	Constructor & Description
1	**Checkbox()** Creates a check box with an empty string for its label.
2	**Checkbox(String label)** Creates a check box with the specified label.
3	**Checkbox(String label, boolean state)** Creates a check box with the specified label and sets the specified state.
4	**Checkbox(String label, boolean state, CheckboxGroup group)** Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group.
5	**Checkbox(String label, CheckboxGroup group, boolean state)** Creates a check box with the specified label, in the specified check box group, and set to the specified state.

Class methods

S.N.	Method & Description
1	**void addItemListener(ItemListener l)** Adds the specified item listener to receive item events from this check box.
2	**void addNotify()** Creates the peer of the Checkbox.
3	**AccessibleContext getAccessibleContext()** Gets the AccessibleContext associated with this Checkbox.
4	**CheckboxGroup getCheckboxGroup()** Determines this check box's group.
5	**ItemListener[] getItemListeners()** Returns an array of all the item listeners registered on this checkbox.
6	**String getLabel()** Gets the label of this check box.
7	**<T extends EventListener>T[] getListeners(Class<T> listenerType)** Returns an array of all the objects currently registered as FooListeners upon this Checkbox.
8	**Object[] getSelectedObjects()** Returns an array (length 1) containing the checkbox label or null if the checkbox is not selected.
9	**boolean getState()** Determines whether this check box is in the **on** or **off** state.
10	**protected String paramString()**

	Returns a string representing the state of this Checkbox.
11	**protected void processEvent(AWTEvent e)** Processes events on this check box.
12	**protected void processItemEvent(ItemEvent e)** Processes item events occurring on this check box by dispatching them to any registered ItemListener objects.
13	**void removeItemListener(ItemListener l)** Removes the specified item listener so that the item listener no longer receives item events from this check box.
14	**void setCheckboxGroup(CheckboxGroup g)** Sets this check box's group to the specified check box group.
15	**void setLabel(String label)** Sets this check box's label to be the string argument.
16	**void setState(boolean state)** Sets the state of this check box to the specified state.

AWT CheckBoxGroup Class

Introduction

The CheckboxGroup class is used to group the set of checkbox.

Class declaration

Following is the declaration for **java.awt.CheckboxGroup** class:

```
public class CheckboxGroup
  extends Object
    implements Serializable
```

Class constructors

S.N.	Constructor & Description
1	**CheckboxGroup() ()** Creates a new instance of CheckboxGroup.

Class methods

S.N.	Method & Description
1	**Checkbox getCurrent()** Deprecated. As of JDK version 1.1, replaced by getSelectedCheckbox().
2	**Checkbox getSelectedCheckbox()** Gets the current choice from this check box group.
3	**void setCurrent(Checkbox box)** Deprecated. As of JDK version 1.1, replaced by setSelectedCheckbox(Checkbox).
4	**void setSelectedCheckbox(Checkbox box)** Sets the currently selected check box in this group to be the specified check box.
5	**String toString()** Returns a string representation of this check box group, including the value of its current selection.

AWT List Class

Introduction

The List represents a list of text items. The list can be configured to that user can choose either one item or multiple items.

Class declaration

Following is the declaration for **java.awt.List** class:

public class List
extends Component
implements ItemSelectable, Accessible

Class constructors

S.N.	Constructor & Description
1	**List()** Creates a new scrolling list.
2	**List(int rows)** Creates a new scrolling list initialized with the specified number of visible lines.
3	**List(int rows, boolean multipleMode)** Creates a new scrolling list initialized to display the specified number of rows.

Class methods

<T extends EventListener> T[] getListeners(Class<T> listenerType)
Returns an array of all the objects currently registered as FooListeners upon this List.

S.N.	Method & Description
1	**void add(String item)** Adds the specified item to the end of scrolling list.
2	**void add(String item, int index)** Adds the specified item to the the scrolling list at the position indicated by the index.
3	**void addActionListener(ActionListener l)** Adds the specified action listener to receive action events from this list.
4	**void addItem(String item)** Deprecated. replaced by add(String).
5	**void addItem(String item, int index)** Deprecated. replaced by add(String, int).
6	**void addItemListener(ItemListener l)** Adds the specified item listener to receive item events from this list.
7	**void addNotify()** Creates the peer for the list.
8	**boolean allowsMultipleSelections()** Deprecated. As of JDK version 1.1, replaced by isMultipleMode().
9	**void clear()** Deprecated. As of JDK version 1.1, replaced by removeAll().
10	**int countItems()** Deprecated. As of JDK version 1.1, replaced by getItemCount().

AWT TextField Class

Introduction

The textField component allows the user to edit single line of text.When the user types a key in the text field the event is sent to the TextField. The key event may be key pressed, Key released or key typed. The key event is passed to the registered KeyListener. It is also possible to for an ActionEvent if the ActionEvent is enabled on the textfield then ActionEvent may be fired by pressing the return key.

Class declaration

Following is the declaration for **java.awt.TextField** class:

public class TextField
extends TextComponent

Class constructors

S.N.	Constructor & Description
1	**TextField()** Constructs a new text field.
2	**TextField(int columns)** Constructs a new empty text field with the specified number of columns.
3	**TextField(String text)** Constructs a new text field initialized with the specified text.
4	**TextField(String text, int columns)** Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

Class methods

S.N.	Method & Description
1	**void addActionListener(ActionListener l)** Adds the specified action listener to receive action events from this text field.
2	**void addNotify()** Creates the TextField's peer.
3	**boolean echoCharIsSet()** Indicates whether or not this text field has a character set for echoing.
4	**AccessibleContext getAccessibleContext()** Gets the AccessibleContext associated with this TextField.
5	**ActionListener[] getActionListeners()** Returns an array of all the action listeners registered on this textfield.
6	**int getColumns()** Gets the number of columns in this text field.
7	**char getEchoChar()** Gets the character that is to be used for echoing.
8	**<T extends EventListener> T[] getListeners(Class<T> listenerType)** Returns an array of all the objects currently registered as FooListeners upon this TextField.
9	**Dimension getMinimumSize()** Gets the minumum dimensions for this text field.

AWT TextArea Class

Introduction

The TextArea control in AWT provide us multiline editor area. The user can type here as much as he wants. When the text in the text area become larger than the viewable area the scroll bar is automatically appears which help us to scroll the text up & down and right & left.

Class declaration

Following is the declaration for **java.awt.TextArea** class:

```
public class TextArea
  extends TextComponent
```

Field

Following are the fields for **java.awt.TextArea** class:

- **static int SCROLLBARS_BOTH** -- Create and display both vertical and horizontal scrollbars.
- **static int SCROLLBARS_HORIZONTAL_ONLY** -- Create and display horizontal scrollbar only.
- **static int SCROLLBARS_NONE** -- Do not create or display any scrollbars for the text area.
- **static int SCROLLBARS_VERTICAL_ONLY** -- Create and display vertical scrollbar only.

Class constructors

S.N.	Constructor & Description
1	**TextArea()** Constructs a new text area with the empty string as text.
2	**TextArea(int rows, int columns)** Constructs a new text area with the specified number of rows and columns and the empty string as text.
3	**TextArea(String text)** Constructs a new text area with the specified text.
4	**TextArea(String text, int rows, int columns)** Constructs a new text area with the specified text, and with the specified number of rows and columns.
5	**TextArea(String text, int rows, int columns, int scrollbars)** Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified.

Class methods

S.N.	Method & Description
1	**void addNotify()** Creates the TextArea's peer.
2	**void append(String str)** Appends the given text to the text area's current text.
3	**void appendText(String str)** Deprecated. As of JDK version 1.1, replaced by append(String).
4	**AccessibleContext getAccessibleContext()** Returns the AccessibleContext associated with this TextArea.
5	**int getColumns()** Returns the number of columns in this text area.
6	**Dimension getMinimumSize()** Determines the minimum size of this text area.
7	**Dimension getMinimumSize(int rows, int columns)** Determines the minimum size of a text area with the specified number of rows and columns.
8	**Dimension getPreferredSize()** Determines the preferred size of this text area.

AWT Choice Class

Introduction

Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

Class declaration

Following is the declaration for **java.awt.Choice** class:

```
public class Choice
  extends Component    implements ItemSelectable, Accessible
```

Class constructors

S.N.	Constructor & Description
1	**Choice() ()** Creates a new choice menu.

Class methods

S.N.	Method & Description
1	**void add(String item)**

	Adds an item to this Choice menu.
2	**void addItem(String item)** Obsolete as of Java 2 platform v1.1.
3	**void addItemListener(ItemListener l)** Adds the specified item listener to receive item events from this Choice menu.
4	**void addNotify()** Creates the Choice's peer.
5	**int countItems()** Deprecated. As of JDK version 1.1, replaced by getItemCount().
6	**AccessibleContext getAccessibleContext()** Gets the AccessibleContext associated with this Choice.
7	**String getItem(int index)** Gets the string at the specified index in this Choice menu.
8	**int getItemCount()** Returns the number of items in this Choice menu.

//java prg for an example of Label,Button,TextField,TextArea,Checkbox
//CheckboxGroup,Choioce and List..

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
        <applet code=awtcontrolsex width=300 height=600>
        </applet>
*/
public class awtcontrolsex extends Applet implements ActionListener
{
        Label lname,lpass,lcity,lgender,laddr,lmarital,lselect,l1,l2,l3;
        TextField name,pass;
        TextArea addr;
        Checkbox male,female,married;
        CheckboxGroup gender;
        Choice city;
        List sel1,sel2;
        Button click;
        public void init()
        {
                lname=new Label("Name: ");
                lpass=new Label("Password: ");
                laddr=new Label("Address: ");
                lcity=new Label("City:  ");
                lgender=new Label("Gender:  ");
                lmarital=new Label("Marital Status:  ");
                lselect=new Label("Selection is:  ");
                l1=new Label(); //empty label...
                l2=new Label();
                l3=new Label();
                name=new TextField(10);
                pass=new TextField(10);
                pass.setEchoChar('#');//password character...
```

```
            addr=new TextArea("",3,20,TextArea.SCROLLBARS_BOTH);
            gender=new CheckboxGroup();
            male=new Checkbox("Male",gender,true);
            female=new Checkbox("feMale",gender,false);
            married=new Checkbox("Married:");
            click=new Button("CLICK ME");
            city=new Choice();
            city.add("Ahmedabad");
            city.add("Rajkot");
            city.add("Jamnagar");
            sel1=new List();
            sel2=new List();
            //set grid layout.....
            setLayout(new GridLayout(12,2));
            add(lname);
            add(name);
            add(lpass);
            add(pass);
            add(laddr);
            add(addr);
            add(lcity);
            add(city);
            add(lgender);
            add(male);
            add(l1);
            add(female);
            add(lmarital);
            add(married);
            add(click);
            add(l2);
            add(lselect);
            add(sel1);
            add(l3);
            add(sel2);
            click.addActionListener(this);
        }
        public void actionPerformed(ActionEvent obj)
        {
            sel1.add("Name : "+name.getText());
            sel1.add("Password: "+pass.getText());
            sel1.add("Address: "+addr.getText());
            sel2.add("City: "+city.getSelectedItem());
            sel2.add("Gender: "+gender.getSelectedCheckbox().getLabel());
            String s=married.getState()?"Married":"Unmarried";
            sel2.add("Marital Status: "+s);
        }
}
```