History of Java:

Java is a general purpose, class based, object oriented, platform independent, portable, architecturally neutral, multithreaded, dynamic, distributed, and robust interpreted programming language.

In 1990, Sun Microsystems has started a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called Stealth Project but later its name was changed to Green Project.

In 1991, James Gosling, Bill Joy, Patric Naughton, Mike Sheradin and several others met in Aspen, Colorado to discuss this project. Mike Sheradin was to focus on business development; Patric Naughton was to begin work on the Graphics system; and James Gosling was to identify proper programming language for the project. Gosling thought C and C++ could be used to develop the project. But the problem he faced with them is that they were system dependent languages and hence could not be used on various processors, which the electronic devices might use. So he started developing a new language, which was completely system independent. This language was initially called Oak. Since the name was registered by some other company, later it was changed to Java.

Why the name Java?: James Gosling and his team members were consuming lot of coffee while developing this language. They felt that they were able to develop a better language because of the good quality coffee they consumed and quality coffee was exported to the entire world from a place called "Java Island". Hence they fixed the name of the place for the language as Java. And the symbol for the Java language is coffee cup and saucer.

By September of 1994, Patric Naughton and others started writing WebRunner- a Java based Web Browser, which was later renamed as HotJava. HotJava was the first browser, having the capabilities of executing Applets, which are programs to run dynamically on Internet. Finally, Sun Microsystems formally announced Java and HotJava in 1995.

Java Version History:

1. **JDK Alpha and Beta (1995)**
2. **JDK 1.0 (23rd Jan, 1996)**
3. **JDK 1.1 (19th Feb, 1997)**
4. **J2SE 1.2 (8th Dec, 1998)**
5. **J2SE 1.3 (8th May, 2000)**
6. **J2SE 1.4 (6th Feb, 2002)**
7. **J2SE 5.0 (30th Sep, 2004)**
8. **Java SE 6 (11th Dec, 2006)**
9. **Java SE 7 (28th July, 2011)**
10. **Java SE 8 (18th March, 2014) - 217 Packages and 4240 Classes.**

Importance of Java to Internet:

The Internet helped Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content.

1. Java Applets: An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server.

2. Security: When we download data from the Internet, there is chance of risks (such as virus, worms, trojans) .For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet

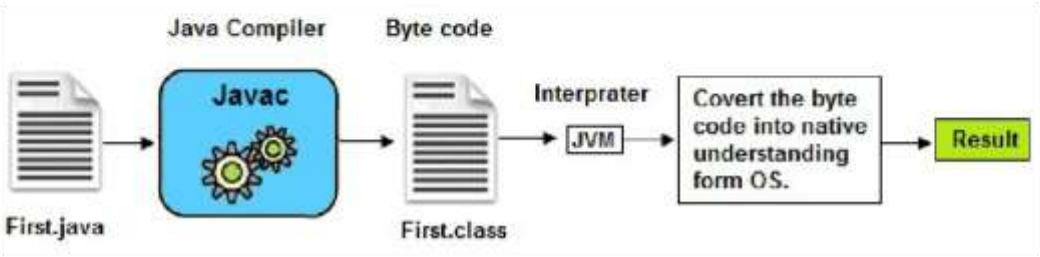
to the Java execution environment and not allowing it access to other parts of the computer.

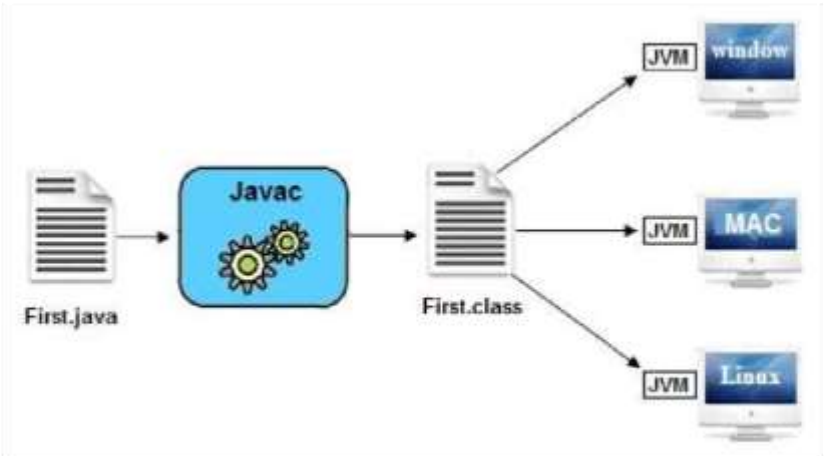
3. Portability: If anyy language supports platform independent annd architectural neutral features, then that langguage is portable and if a program yields the same result on any machine, then that proggram is also called portable. Java achieves thiis with byte code.

Java's Magic: The Bytecode

Java solves both the security and the portability problem is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).

In essence, the original JVM was designed as an interpreter for bytecode. Translating a Java program into bytecodee makes it much easier to run a program in a wide variety of environments because only thhe JVM needs to be implemented for each plaatform.

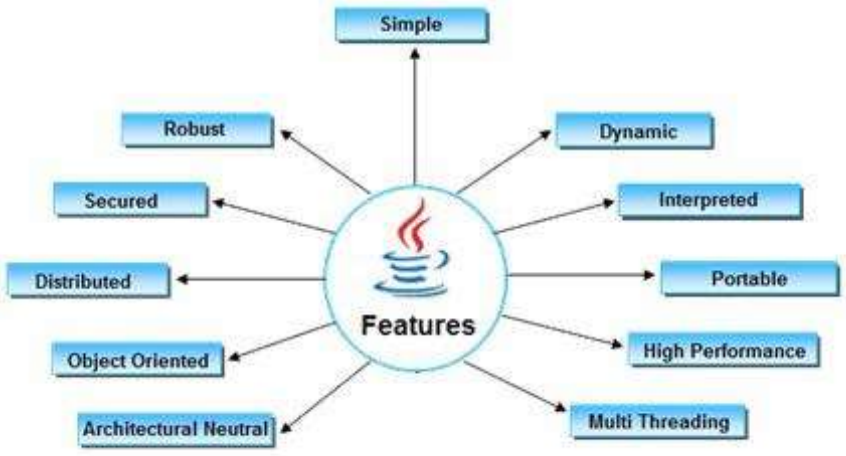




In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect. Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was by Java people. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted.

Features of Java/ Java Buzzwords:

Features of Java are:



1. Simple:

I. It is free from pointer due to this execution time of application is improve.

II. It have rich set of APIs (application programming interface).

III. It have Garbage Collector which is always used to collect un-referenced (unused) memory locations for improving performance of a Java program.

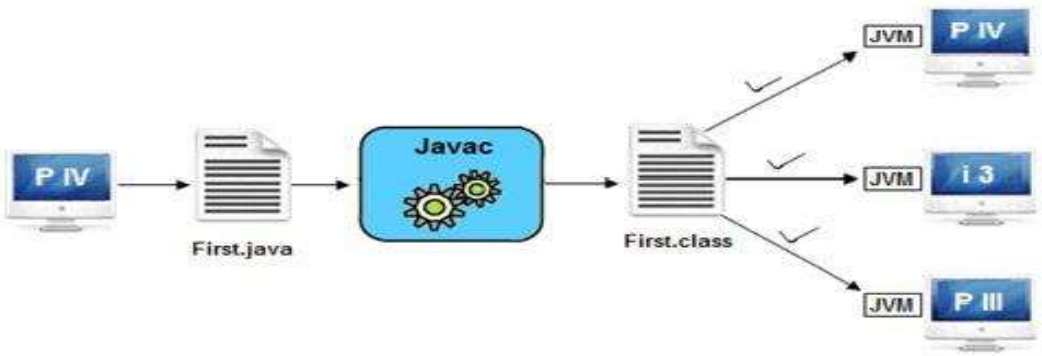
IV. It contains user friendly syntax for developing any applications.

2. Object Oriented: Java supports OOP features and everything is an Object in Java. The object model in Java is simple and easy to extend.

3. Robust: Java is robust or strong programming language because of its capability to handle run-time errors, automatic garbage collection, lack of pointer concept, exception handling. These entire things make Java is a robust Language.

4. Multi Threading: Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

5. Architecture-Neutral: Architecture represents processor. A language or technology is said to be architectural neutral which can run on any processors in the real world without considering type of architecture and vendor (providers).



6. **Interpreted: Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter in JVM. If we consider any other language, only an interpreter or a compiler to execute the programs. But in Java, we use both interpreter and compiler for the execution.**

7. **High Performance: The problem with interpreter inside JVM is that it is slow. Because of this, Java programs used to run slow. To overcome this, along with interpreter, Java people have introduced JIT (Just-in-Time) compiler, which enhances the speed of execution. So now in JVM, both interpreter and JIT compiler works together to run the program.**

8. **Distributed: Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.**

9. **Dynamic: Java support dynamic memory allocation. Due to this memory wastage is reduced. The process of allocating the memory space to the program at a run-time is known as dynamic memory allocation, To allocate memory space dynamically we use an operator called 'new'. 'new' operator is known as dynamic memory allocation operator.**

10. **Secure: It is more secured language compare to other languages; In this, program is covered into byte code after compilation which is not readable by human and also Java does not support global variable concept.**

11. **Portable: If any language supports platform independent and architectural neutral features, then that language is portable.**

Portable = Platform independent + Architecture-neutral

Differences between Java and C++:

S. No	Java	C++
1	Java is a pure object oriented programming language; therefore, everything is an object in Java	C++ supports both procedural and object oriented programming; therefore, it is called a hybrid language.
2	Java does not support pointers, templates, unions, operator overloading, structures etc.	C+ supports structures, unions, templates, operator overloading, pointers and pointer arithmetic.
3	Java support automatic garbage collection. It does not support destructors as C++ does.	C++ support destructors, which is automatically invoked when the object is destroyed.
4	Java has built in support for threads. In Java, there is a Thread class that you inherit to create a new thread.	C++ has no built in support for threads. C++ relies on non-standard third-party libraries for thread support.
5	There is no scope resolution operator (::) in Java. The method definitions must always occur within a class.	C++ has scope resolution operator (::) which is used to define a method outside a class and to access a global variable within from the scope where a local variable also exists with the same name.
6	There is no goto statement in Java. The keywords const and goto are reserved, even though they are not used.	C++ has goto statement.
7	Java doesn't provide multiple inheritance	C++ support multiple inheritance. The keyword virtual is used to resolve ambiguities during multiple inheritance if there is any.
8	Java is interpreted for the most part and hence platform independent.	C++ generates object code and the same code may not run on different platforms.

Java is a strongly typed language:

In Java, every variable has a type, every expression has a type, and every type is strictly defined. And, also all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.

The Primitive Types:

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types.

These can be put in four groups:

1. **Integers: This group includes byte, short, int, and long, which are for whole-valued signed numbers. Following table specified its width and range.**

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

2. **Floating-point numbers: This group includes float and double, which represent numbers with fractional precision.**

Name	Width	Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

3. **Characters: This group includes char, which represents symbols in a character set, like letters and numbers. In Java, character data type occupies two bytes of memory because of UNICODE representation.**

4. **Boolean: This group includes boolean, which is a special type for representing true/false values.**

   Ex: boolean b;

Variables: Variable represents basic unit of storage in a Java program. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [ = value ][, identifier [= value ] …];
```

Example:

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three  ints, initializing d and f.
byte z = 22; // initializes z.
```

/* Java program to illustrate boolean data type*/

```
class Bool_Example
{
    public static void main(String args[])
    {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if
        statement if(b) System.out.println("This
        is executed."); b = false;
        if(b) System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean
        value System.out.println("10 > 9 is " + (10 > 9));
    }
}
```


.

Dynamic Initialization:

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic
initialization. class DynInit {
        public static void main(String args[]) {
                double a = 3.0, b = 4.0;
            // c is dynamically initialized
            double c = Math.sqrt(a * a + b * b);
            System.out.println("Hypotenuse is " + c);
        }
}
```

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse.

Scope and lifetime of variables:

Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: global and local. In Java, the two major scopes are those defined by a class and those defined by a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.

.

/* Write a Java program to illustrate scope of a variable*/

```
// Demonstrate block scope.
class Scope_Example
{
        public static void main(String args[])
        {
                int x; // known to all code within
                main x = 10;
                if(x == 10)
                {// start new scope
                        int y = 20; // known only to this block // x
                        and y both known here.
                        System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                }
                // y = 100; // Error! y not known here
                // x is still known here.
                System.out.println("x is " + x );
        }
}
```

Type Conversion and Casting: Assigning a value of one type to a variable of another type.

Java supports two types of type casting.

1. **Automatic type casting:**

   Automatic type conversion will take place if the following two conditions are met:

   - **The two types are compatible.**
   - **The destination type is larger than the source type.**

   When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

2. **Explicit type casting: 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:**

   variable=(target-type) value

/* Java program to illustrate explicit type casting*/

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

```
        {
                byte b;
                int i = 257;
                double d = 323.142;
                System.out.println("\nConversion of int to byte.");
                b = (byte) i;
                System.out.println("i and b " + i + " " + b);
                System.out.println("\nConversion of double
                to int."); i = (int) d;
                System.out.println("d and i " + d + " " + i);
                System.out.println("\nConversion of double to
                byte."); b = (byte) d;
                System.out.println("d and b " + d + " " + b);
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 1>java Conversion_Example

Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67

D:\Materials\JAVA Material\Unit 1>
```

Arrays:

- An array is a group of variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.

One-Dimensional Arrays:

The general form of a one-dimensional array declaration is

type var-name[ ];

Ex: int month_days[];

Although this declaration establishes the fact that month_days is an array variable, no array actually exists. In fact, the value of month_days is set to null, which represents an array with no

value. To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days. new is a special operator that allocates memory.

var_name = new type [size];

month_days = new int[12];

- Another version:

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

int month_days[] = new int[12];

- Another version:
  - int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; Here, we can initialize arrays in curly braces by separating with comma.

Alternative Array Declaration Syntax:

- There is a second form that may be used to declare an array: type[ ] var-name;
- Here, the square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

- The following declarations are also equivalent: char twod1[][] = new char[3][4];

char[][] twod2 = new char[3][4];

- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,
  - int[] nums, nums2, nums3; // create three arrays creates three array variables of type int.

It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```

```
/*  Java program to illustrate arrays*/
class Array_Example
{
        public static void main(String args[])
        {
                int month_days[];
                month_days = new int[12];
                month_days[0] = 31;
                month_days[1] = 28;
                month_days[2] = 31;
                month_days[3] = 30;
                month_days[4] = 31;
                month_days[5] = 30;

                month_days[7] = 31;
                month_days[8] = 30;
                month_days[9] = 31;
                month_days[10] = 30;
                month_days[11] = 31;
                System.out.println("April has " + month_days[3]+ " days.");
        }
}
```

Multi-Dimension arrays:

To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two dimensional

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Operators:

1. **Arithmetic : +, -,*,/,%,++,+=,-=,*=,/=,%=,--**
2. **Bitwise operators(works on only integer group): &, |, ^, >>,<<, >>>,<<<, &=, |=, ^=**
3. **Relational: ==,!=, >,<,>=,<=**
4. **Boolean logical operands(Works only on boolean operands): &, |, ^, ||, &&, !, !=, ?:, ==**

Control Statements:

Java's program control statements can be put into the following categories: selection, iteration, and jump.

- **Selection statements: allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.**
  - **if and its variants, switch**
- **Iteration statements enable program execution to repeat one or more statements until some condition is satisfied.**
  - **while, do-while, for, for-each**
- **Jump statements allow your program to execute in a nonlinear fashion.**
  - **break, continue, return**

Selection statements: Java supports two selection statements:

1. **if**
2. **switch**

if statement:

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition)
        statement1;
else
        statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

Nested if:

A nested if is an if statement that is the target of another if or else. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
        if(j < 20) a = b;
        if(k > 100) c = d; // this if is
        else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) because it is the closest if within the same block.

The if-else-if Ladder: A common programming construct that is based upon a sequence of nested ifs is the if-elseif ladder. It looks like this:

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
          .
          .
          .
else
        statement;
```

```
/* Java program to demonstrate the use of if
statement*/ class CheckNumber{
    public static void main(String args[]){
        int num = 10;
        if (num % 2 == 0){
            System.out.println(num + " is even number );
        }
        else {
            System.out.println(num + " is odd number );
        }     } }
```

Switch-case statement: switch-case can be used as an alternative for if-else condition. If the programmer has to make number of decisions, and all the decisions depend on the value of variable, then switch-case is used instead of if-else condition.

Syntanx of switch-case:

```
switch(expression){
        case 1 :
              action1 statement;
              break;
        case 2:
              action2 statement;
              break;
                    .
                    .
                    .
    case N:
              actionN statement;
                break;
        default:
              default statement;
}
```

Where expression: is variable containing the value to be evaluated. It must be of type byte, short, int, char and enumeration. From Java 7 onwards string types are also allowed. default: is an optional keyword used to specify the statements to be executed only when all the case statements evaluate to false.

```
/* Java program to demonstrate the use of switch statement*/
class StringSwitch_Example
{
    public static void main(String args[])
    {
        String str = "two";
        switch(str) {
            case "one":
                System.out.println("one");
                break;
            case "two":
                System.out.println("two");
                break;
            case "three":
                System.out.println("three");
                break;
            default:
```

```
                    System.out.println("no match");
                    break;
                }
            }
        }
```

Nested switch statement: switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.

For example, the following fragment is perfectly valid:

```
switch(count) {
        case 1:
            switch(target) { // nested switch
                case 0:
                    System.out.println("target is zero");
                    break;
                case 1: // no conflicts with outer switch
                    System.out.println("target is one");
                    break;
            }
            break;
            case 2: // ...
            .
```

Iteration Statements:

1. **While loop: The while loop executes till condition is specified. It executes the steps mentioned in the loop only if the condition is true.**

Syntanx of while loop:

```
while (condition) {
        statements;
        statements;
    }
```

Where condition, is any boolean expression that returns a true or false value. The loop continues upto condition returns true value.

Example :

```
int i = 1;
while(i<=5){
        i++;
        System.out.println(“value of i is : “+i);
}
```

2. **do-while loop: The do-while loops execute certain statements till the specified condition is true. This loop ensures that the loop body is executed at least once.**

Syntax of do-while

```
loop: do{
        action statements;
}while(condition);
```

Example :

```
int j = 8;
do{
    j = j+2;
    System.out.println("value of j is : " +j);
}while(j>=10 && j<=50);
```

3. **for loop: All loops have some common feature: a counter variable that is initialized before the loop begins. The for loop provides the feature that, initialized the variable before loop begins, a condition for counter variable and counter upto which loop lasts.**

Syntax of for loop:

```
for(initialization statements; condition; increment/decrement
   statements){ action statements;
            .
            .
}
```

where initialization statements : sets or initialize the value for counter variable.

condition : A boolean expression that returns true or false value. The loop terminates if false value is returned.

Increment/decrement statements : Modifies the counter variable. These statements are always executed after the action statements, and before the subsequent condition check.

Example:

```
for(int i=1;i<=10;i++){
        System.out.println("Value of i is " +i);
}
```

4. **for-each: The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.**
   It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

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

Example:

```
public class ForEachExample
{
 public static void main(String[] args) {
        int arr[]={12,23,44,56,78};
        for(int i:arr)
        {
            System.out.println(i);
        }
    }
 }
```

Jump statements:

1. **break statement: In Java, the break statement has three uses.**
   - **First, it terminates a statement sequence in a switch statement.**
   - **Second, it can be used to exit a loop.**
   - **Third, it can be used as alternative to goto statement.**
     - **Syntax:**
       - **break label;**

```
/* Java program to demonstrate the use of brake statement*/
class BreakLoop
{
        public static void main(String args[])
        {
                for(int i=0; i<100; i++)
                {
                        if(i == 10) break; // terminate loop if i is 10
                                System.out.println("i: " + i);
                }
                System.out.println("Loop complete.");
        }
}
```

```
/* Java program to demonstrate the use of brake statement in
    labels */ class BreakStatement
    {
        public static void main(String args[])
        {
            boolean t = true;
            first:
            {
                second:
                {
                    third:
                    {
                        System.out.println("Before the break.");
                        if(t)
                            break second; // break out of second block
                         System.out.println("This won't execute");
                    }
                    System.out.println("This won't execute");
                }
                System.out.println("This is after second block.");
            }
        }
    }
```

Output:

```
D:\Materials\JAVA Material\Unit 1>java BreakStatement
Before the break.
This is after second block.
```

2. **continue statement:** Sometimes the programmer might want to continue a loop, but stop processing the remainder of the code in its body for a particular iteration. The continue statement can be used for such a scenario. It can be also used in Java as alternative to goto statement.

```
/* Java program to demonstrate the use of continue statement*/
    class ContinueStatement
    {
        public static void main(String args[])
        {
            for(int i=0; i<10; i++)
            {
                if (i==5)
                    continue;
                System.out.println(i); } } }
```

```
D:\Materia
0
1
2
3
4
6
7
8
9
```

```
/* Java program to demonstrate the use of continue with labels */
import java.util.*;
public class Factorial
{
    public static void main(String[] args)
    {
        int x = 1;
        int factValue = 1;
        Scanner userInput = new Scanner(System.in);
        restart:
        while(true)
        {
            System.out.println("Please enter a nonzero,
            nonnegative "); int factInput = userInput.nextInt();
            if(factInput<=0)
            {
                System.out.println("Enter a nonzero, nonnegative
                value "); factInput = userInput.nextInt();
            }
            if(factInput<1)
            {
                System.out.println("The number you entered is not
                                    valid. Please try again.");
                continue restart;
            }
            while(x<=factInput)
            {
                factValue*=x;
                x++;
            }
            System.out.println(factInput+"! = "+factValue);
            break restart;
        }
    }
}
```

3. **Return statement:**

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

The following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main( ):

```
class Return
{
        public static void main(String args[])
        {
                boolean t = true;
                System.out.println("Before the return.");
                if(t)
                        return; // return to caller
                System.out.println("This won't execute.");
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 1>java Return
Before the return.
```

Scanner Class:

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program.

1. To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
2. To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
3. To read strings, we use nextLine().
4. To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

Methods:

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

```
/* Java program to demonstrate the use of scanner class*/
import java.util.Scanner;
public class ScannerDemo1
{
   public static void main(String[] args)
   {
            Scanner sc = new Scanner(System.in);
     System.out.println("Enter Name");
     String name = sc.nextLine();

     System.out.println("Enter Male/Female - M/F");
              char gender = sc.next().charAt(0);

             System.out.println("Enter Age, Mobile Number and
             CGPA"); int age = sc.nextInt();
              long mobileNo = sc.nextLong();
              double cgpa = sc.nextDouble();

       // Print the values to check if input was correctly
       obtained. System.out.println("Name: "+name);
       System.out.println("Gender: "+gender);
       System.out.println("Age: "+age);
       System.out.println("Mobile Number: "+mobileNo);
       System.out.println("CGPA: "+cgpa);
   }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 1>java ScannerDemo1
Enter Name
Saadvitha
Enter Male/Female - M/F
F
Enter Age, Mobile Number and CGPA
3
9490000000
9.21
Name: Saadvitha
Gender: F
Age: 3
Mobile Number: 9490000000
CGPA: 9.21

D:\Materials\JAVA Material\Unit 1>
```

Java Strings:

In java, string is basically an object that represents sequence of char values and string objects are immutable (cannot be modified).

Crating Strings: There are three ways to create strings in Java.

1. Create a string just by assigning a group of characters to a string type variable.

   String s; // declare a string type variable

   s=”PVPSIT”; //assign a group of characters to it

   Above two statements can be combined into

   String s= “PVPSIT”;

   In this case JVM creates an object to String class and stores the string “PVPSIT” in that object and is referenced with the name s.

2. We can create an object to String class by allocating memory using new operator. This is similar to object creating an object to any class.

   String s= new String(“PVPSIT”);

3. Another way of creating strings is by converting character arrays into

   strings. char arr[]={ ‘P’, ‘V’,’P’,’S’,’I’,’T’};

   Now, create a string object by passing the array name to it as

   String s=new String(arr);

   This means all the characters of array are copied into the string s. If we don’t want all characters of the array into the array, then we can mention which characters we need:

   String s=new String(arr, indice_location, size);

   String s=new String(arr,2,3);

String Class Methods:

Sl. No	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex-1)	returns substring for given begin index and end index
7	boolean equals(Object another)	checks the equality of string with object
8	boolean isEmpty()	checks if string is empty

9	String concat(String str)	concatenates specified string
10	String equalsIgnoreCase(String another)	Compares another string. It doesn't check case.
11	String[] split(String regex)	returns splitted string matching regex
12	int indexOf(int ch)	returns specified char value index
13	int indexOf(String substring)	returns specified substring index
14	String toLowerCase()	returns string in lowercase.
15	String toUpperCase()	returns string in uppercase.
16	String trim()	removes beginning and ending spaces of this string.
17	int compareTo(String str)	is used for comparing two strings lexicographically. Each character of both the strings is converted into a Unicode value for comparison. If both the strings are equal then this method returns 0 else it returns positive or negative value.
18	int compareToIgnoreCase (String str)	comparison of two strings irrespective of cases.

/* Java program to demonstration of String methods*/

```
class String_Demo
{
	public static void main(String arg[])
	{
		String s1="PVP Siddhartha";
		String s2=new String("Engineering");
		String s3=new String("College");

		System.out.println("First	:"+s1);
		System.out.println("Second	:"+s2);


		System.out.println("Length of first String			:"+s1.length());
		System.out.println("Concatenation of first and second	:"+s1.concat(s2));

		System.out.println("Concatenation of strings with + :"+s1+" "+s3);
		System.out.println("If string s1 starts with P or not	:"+s1.startsWith("P"));

		System.out.println("Extraction of substrings		:"+s1.substring(2));
		System.out.println("Extraction of substrings		:"+s1.substring(2,5));

		System.out.println("String into uppercase		:"+s1.toUpperCase());
```

```
                System.out.println("String into lower case          :"+s1.toLowerCase());

                String str = " Hello ";
                System.out.println("Use of Trim()                   :"+str.trim());
        }
}
```

/*Use of split() method*/

```
class String_Split
{
        public static void main(String ar[])
        {
                String str="PVP Siddhartha Inst. of Technology";

                //declare a string type array s to store pieces
                String s[];

                s=str.split(" "); //space

                for(int i=0;i<s.length;i++)
                        System.out.println(s[i]);
        }
}
```

Comparison of two strings: Relational operators <,<=,>,>=,==, != cannot be used to compare two strings. On the other hand, methods like compareTo(), compare() can be used.

```
                /* Comparison of two strings using == operator */

class CompareOperator
{
        public static void main(String arg[])
        {
                String s1="Hello";
                String s2=new String("Hello");

                if(s1==s2)
                        System.out.println("Same");
                else
                        System.out.println("Not Same");
        }
}
```

Output: Not Same

Explanation: When an object is created by JVM, it returns the memory address of the object as a hexadecimal number, which is called object reference. Whenever, a new object is

.

created, a new reference numbers is allotted to it. It means every object will have a unique reference.

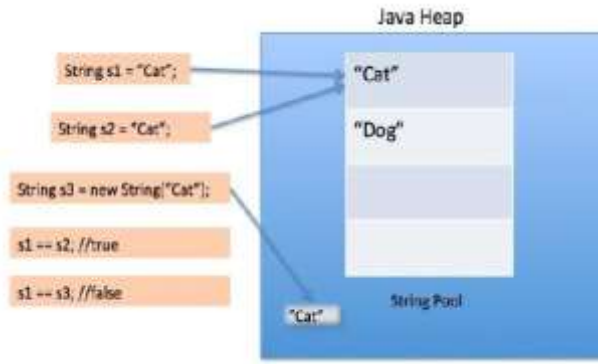
So, to compare two strings use **equals()** method.

```
class CompareEqual
{
	public static void main(String arg[])
	{
		String s1="Hello";
		String s2=new String("Hello");

		if(s1.equals(s2))
			System.out.println("true");
		else
			System.out.println("false");

	}
}
```

Output: true

Difference between == and equals() while comparing strings: == operator compares the reference of the string objects, not the content of the objects. Where as equals method compares the contents of the strings not object references.

String pool:




```
class String_Pool
{
	public static void main(String arg[])
	{
		String s1="Cat";
		String s2= "Cat";

		if(s1==s2)
			System.out.println("true");
		else
			System.out.println("false");

	}
}
```

Here, JVM creates a String object and stores “Welcome” in it. Observe that we are not using new operator to create the string. We are using = operator for this purpose. So after creating the String object, JVM uses a separate block of memory which is called **string constant pool** and stores the object**.**

In next statement when, s2 is executed by the JVM, it searches in the string constant pool to know whether the object with same content is already available or not. If so, copies the reference of that object to s2. So, we have same value in s1 and s2.

StringBuffer Class:

A String class objects are immutable and hence their contents cannot be modified. StringBuffer class objects are mutable, so they can be modified. Moreover the methods that directly manipulate the data of the object are not available in String class. Such methods are available in StringBuffer class.

Creating StringBuffer objects:

There are two ways to create StringBuffer object, and fill the object with string.

1. We can create a StringBuffer object by using new operator and pass the string to the object, as:

```
Ex 1.  StringBuffer sb=new StringBuffer("Hello");
Ex 2.  String str=new String("Hello");
       StringBuffer sb=new StringBuffer(str);
```

2. Second way is, allocate the memory to the StringBuffer object with new operator and later store string data into it.

```
StringBuffer sb=new StringBuffer();
```

Here, we are creating a StringBuffer object and empty object and not passing any string to it. In this case, object will be created with default capacity of 16 characters.

```
StringBuffer sb=new StringBuffer(50); //50 characters
sb.append("Hello");
        or
sb.insert(0,"Hello");
```

Methods:

Sl. No	Method	Description
1	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc
2	insert(int offset, String s)	is used to insert the specified string with this string at the specified position.
3	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
4	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex
5	reverse()	is used to reverse the string

6	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
7	charAt(int index)	is used to return the character at the specified position.
8	length()	is used to return the length of the string i.e. total number of characters
9	String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
10	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Example Program:

```
class StringBufferDemo
{
	public static void main(String arg[])
	{
		StringBuffer sb=new StringBuffer();
		System.out.println("Use of capacity():"+sb.capacity());

		StringBuffer sb1=new StringBuffer("Uni");
		System.out.println("Use of append():"+sb1.append("versity"));
		System.out.println("Use of insert():"+sb1.insert(0, "JNTU "));
		System.out.println("use of delet()"+sb1.delete(0,4));
		System.out.println("Use of length():"+sb1.length());

		System.out.println("Use of reverse():"+sb1.reverse());
		System.out.println("Use of capacity():"+sb1.capacity());
		System.out.println("Use of reverse():"+sb1.reverse());
		System.out.println("Use of indexOf():"+sb1.indexOf("i"));
		System.out.println("Use of lastIndexOf:"+sb1.lastIndexOf("i"));
		System.out.println("Use of substring(int):"+sb1.substring(4));
		System.out.println("Use of substring(int,int):"+sb1.substring(0,4));
		System.out.println(sb1);
		System.out.println("Use of replace():"+sb1.replace(0,3,"ANU "));
	}
}
```

Output:

```
Use of capacity():16
Use of append():University
Use of insert():JNTU University
use of delet() University
Use of length():11
Use of reverse():ytisrevinU
Use of capacity():19
Use of reverse(): University
Use of indexOf():3
Use of lastIndexOf:8
Use of substring(int):versity
Use of substring(int,int): Uni
 University
Use of replace():ANU iversity
```

StringTokenizer Class:

The **java.util.StringTokenizer** class allows you to break a string into pieces called tokens. These tokens are then stored in the StringTokenizer object.

The code to create an object to StringTokenizer class is

1. StringTokenizer obj=new StringTokenizer(String str);
   Splits the given string into tokens based on default delimiter space.
2. StringTokenizer obj=new StringTokenizer(String str, String delimter); Splits the given string into tokens based on the delimeter.
   Ex. StringTokenizer obj=new StringTokenizer("pvp,Siddhartha,inst,tech","","");
   Splits the above string into tokens based on the delimeter comma.

Methods:

S. No	Method	Description
1	boolean hasMoreTokens()	checks if there is more tokens available.
2	String nextToken()	returns the next token from the StringTokenizer object.
3	boolean hasMoreElements()	same as hasMoreTokens() method.
4	Object nextElement()	same as nextToken() but its return type is Object.
5	int countTokens()	returns the total number of tokens.

Example:

```
import java.util.StringTokenizer;
class Simple
{
	public static void main(String args[])
	{
		StringTokenizer st = new StringTokenizer("my,name,is,pvpsit",",");
		while (st.hasMoreTokens())
		{
			System.out.println(st.nextToken());
		}
	}
}
```

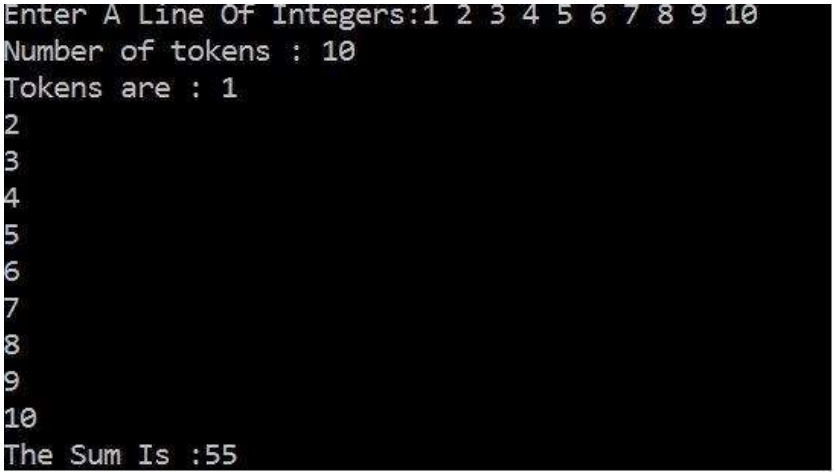
Output:

```
my
nameDifference between spilt and StringTokenizer:
is
pvpsit
```

/* Write a Java Program that reads a line of integers, and then displays each integer, and the sum of all the integers (Use StringTokenizer class of java.util) */

```
import java.util.*;

class StringTokenizerEx
{
   public static void main(String[] args)
   {
        Scanner s = new Scanner(System.in);
      System.out.print("\nEnter A Line Of Integers:");

      String line = s.nextLine();

      StringTokenizer st = new StringTokenizer(line);
      System.out.println("Number of tokens : "+st.countTokens());
      int sum = 0;
      System.out.print("Tokens are : " );
      while (st.hasMoreTokens())
      {
         int i = Integer.parseInt(st.nextToken());
         System.out.println(i);
         sum = sum + i;
      }
      System.out.println("The Sum Is :" +sum);
    }
}
```

Output:



Difference between StringTokenizer and split():

In StringTokenizer, the delimiter is just one character long. You supply a list of characters that count as delimiters, but in that list, each character is a single delimiter. With split(), the delimiter is a regular expression, which is something much more powerful (and more complicated to understand). It can be any length.

```
import java.util.StringTokenizer;
class D_Split_ST
{
	public static void main(String args[])
	{
		StringTokenizer st = new StringTokenizer("This is another example",
		"char"); //delimiters are individual characters such as c, h, a r

		while (st.hasMoreTokens())
		{
			System.out.println(st.nextToken());

		}

		String str="This is another example";

		//declare a string type array s to store pieces
		String s[];

		s=str.split("char"); // here the delimiter is char

		for(int i=0;i<s.length;i++)
			System.out.println(s[i]);

	}
}
```

Output:

```
T
is is
not
e
 ex
mple
This is another example
```

/* Write a Java program that checks whether a given string is a palindrome or not. Ex MALAYALAM is a palindrome. */

```
import java.io.*;
class Palindrome
{
	public static void main(String args[])
	{
		String str1=new String(args[0]);
		StringBuffer str2=new StringBuffer(str1);
		str2.reverse();

		for(int i=0;i<str2.length();i++)
		{
			if(str1.charAt(i)!=str2.charAt(i))
			{
				System.out.println(str1+" is not palindrome");
				System.exit(0);
			}
		}
		System.out.println(str1+" is palindrome");
	}
}
```

/* Write a Java program for sorting a given list of names in ascending order.*/

```
import java.util.*;
class SortEx
{
	public static void main(String arg[ ])
	{
		Scanner s=new Scanner(System.in);

		System.out.println("Enter the size");
		int n=s.nextInt();

		String array[]=new String[n];

		System.out.println("Enter names");
		for(int i=0;i<n;i++)
		{
			array[i]= s.next();
		}

		for(int i=0;i<array.length-1;i++)
		{
			for(int j=i+1;j<array.length;j++)
			{
				if(array[i].compareToIgnoreCase(array[j])>0)
				{
					String Temp = array[i] ;
					array[i] = array[j] ;
```

```
                                        array[j] = Temp ;
                                }
                        }
                }
                System.out.println("After Sorting");
                for(int i=0;i<n;i++)
                {
                        System.out.println(array[i]);
                }
        }
}
```

Inheritance:

- Inheritance can be defined as the process where one class acquires the properties of another class
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
- Inheritance represents the IS-A relationship, also known as parent-child relationship.
- Advantages of inheritance:
    - Code reusability
    - Used in method overriding (so runtime polymorphism can be achieved).

Syntax:

```
class Subclass-name extends Superclass-name
{
        //methods and fields
}
```

The extends keyword indicates that you are making a new class that derives from an existing class.

Example:

```
// 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 [])
        {
                B subOb = new B();

                /* The 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();
                System.out.println("Sum of i, j and k in subOb:");
                subOb.sum();
        }
}
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>java SimpleInheritance
Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

Member Access and Inheritance

Although a subclass includes all of the members of its super class, it cannot access those members of the super class that have been declared as private. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class. This program contains an
error and will not compile. */
        class A
        {
                int i; // public by default
                private int j;  //Private to A
```

```
        void setij(int x, int y)
        {
                i = x;
                j = y;
        }
}
class B extends A
{
        int total;
        void sum()
        {
                total = i + j;  //A's j is not accessible here
        }
}
class SimpleInheritance2
{
        public static void main(String args[])
        {
                B subOb = new B();
                subOb.setij(10,  12);
                subOb.sum();
                System.out.println("Total is " + subOb.total);
        }
}
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac SimpleInheritance2.java
SimpleInheritance2.java:22: error: j has private access in A
                total = i + j;  //A's j is not accessible here
                            ^
1 error
```

A super class variable can reference a subclass object:

A reference variable of a super class can be assigned a reference to any subclass derived from that super class.

Ex:

```
        class A
        {
                void callMe()
                {
                        System.out.print("Hello");
                }
        }
```

```
class B extends A
{
        void callMe()
        {
                System.out.print("Hi ");
        }
 }
class Reference
{
        public static void main(String args[])
        {
                A ref;
                B b = new B();
                ref = b;
                ref.callMe();
        }
}
```

Output: Hi

Using super

Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity. To overcome this, super is used to refer super class properties.

The super keyword in java is a reference variable that is used to refer parent class. The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

- super is used to refer immediate parent class instance variable.
  super.parent_instance_variable_name;
- super is used to invoke immediate parent class method
  super.parent_class_method_name();
- super is used to invoke immediate parent class constructor.
  super(arglist); // parameterized constructor
  super(); //default

Usage 1. - Using super to refer super class property

```
class Vehicle
{
        int speed=50;
}
class Bike extends Vehicle
{
         int speed=100;
         void display()
         {
             System.out.println("Vehicle Speed:"+super.speed);//will print speed of vehicle
             System.out.println("Bike Speed:"+speed);//will print speed of bike
         }
}
class SuperVarible
{
            public static void main(String args[])
           {
                    Bike b=new Bike();
                    b.display();
            }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperVarible
Vehicle Speed:50
Bike Speed:100
```

Usage 2. - super is used to invoke immediate parent class method

```
class Student
{
          void message()
          {
                    System.out.println("Good Morning Sir");
          }
}

class Faculty extends Student
{
```

```
        void message()
        {
                System.out.println("Good Morning Students");
        }
        void display()
        {
                message();//will invoke or call current class message() method
                super.message();//will invoke or call parent class message() method
        }
}
class SuperClassMethod
{
        public static void main(String args[])
        {
                Faculty f=new Faculty();
                f.display();
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperClassMethod
Good Morning Students
Good Morning Sir
```

Usage 3. - super is used to invoke immediate parent class constructor

```
class Vehicle
{
          Vehicle( )
          {
                System.out.println("Vehicle is created");
          }
}

class Bike extends Vehicle
{
           Bike()
           {
                super( );//will invoke parent class constructor
                System.out.println("Bike is created");
           }

}
class SuperClassConst
{
```

```
        public static void main(String args[])
        {
            Bike b=new Bike();
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperClassConst
Vehicle is created
Bike is created
```

Calling Constructors:

Constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}
// Create a subclass by extending class A.
class B extends A
{
    B() {
        System.out.println("Inside B's constructor.");
    }
}
// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}
class CallingConstructor
{
    public static void main(String args[])
    {
```

```
        C c = new C();
    }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java CallingConstructor
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

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 override the method in the superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Example:

```
// 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
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java Override
k: 3
```

Dynamic Method Dispatch:

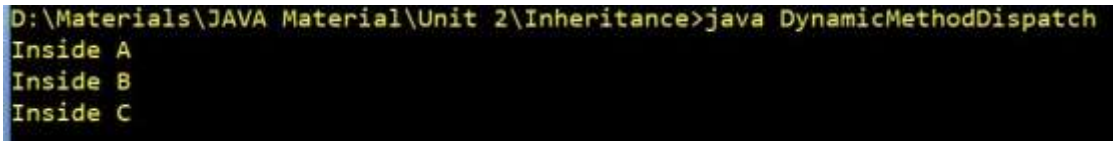
Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Example: //A Superclass Variable Can Reference to a Subclass Object

```
class A
{
        void callMe()
        {
            System.out.println("Inside A");
        }
}
class B extends A
{
        void callMe()
        {
            System.out.println("Inside B");
        }
}

class C extends B
{
```

```
        void callMe()
        {
                System.out.println("Inside C");
        }
}
class DynamicMethodDispatch
{
        public static void main(String args[])
        {
                A a=new A();
                B b = new B();
                C c = new C();
                A ref;
                ref = a;
                ref.callMe();

                ref = b;
                ref.callMe();

                ref=c;
                ref.callMe();
        }
}
```

Output:



Applying Method Overriding:

```
import java.util.*;

class Student
{
        int n;
        String name;
        void read()
        {
                Scanner s=new Scanner(System.in);
                System.out.println("Enter no and Name");
                n=s.nextInt();
                name=s.nextLine();

        }
```

```
	void show()
	{
		System.out.println("No:"+n);
		System.out.println("Name:"+name);
	}
}
class ITStudent extends Student
{
	void read()
	{
		super.read();
	}
	void show()
	{
		super.show();
	}
}
class CSEStudent extends Student
{
	void read()
	{
		super.read();
	}
	void show()
	{
		super.show();
	}

}
class ECEStudent extends Student
{
	void read()
	{
		super.read();
	}
	void show()
	{
		super.show();
	}
}
class MainMethod
{
	public static void main(String ar[])
	{
		ITStudent it=new ITStudent();
		CSEStudent cse=new CSEStudent();
```

```
                ECEStudent ece=new ECEStudent();

                Student sref;

                sref=it;
                sref.read();
                sref.show();

                sref=cse;
                sref.read();
                sref.show();

                sref=ece;
                sref.read();
                sref.show();
        }
}
```

Abstract Class

- An abstract class is a class that contains one or more abstract methods.
- An abstract method is a method without method body.

  Syntax:

  ```
  abstract  return_type method_name(parameter_list);
  ```

- An abstract class can contain instance variables, constructors, concrete methods in addition to abstract methods.
- All the abstract methods of abstract class should be implemented in its sub classes.
- If any abstract method is not implemented in its subclasses, then that sub class must be declared as abstract.
- We cannot create an object to abstract class, but we can create reference of abstract class.
- Also, you cannot declare abstract constructors or abstract static methods.

Example 1: Java program to illustrate abstract class.

```
abstract class MyClass
{
        abstract void calculate(double x);
}
class Sub1 extends MyClass
{
        void calculate(double x)
```

```
        {
                System.out.println("Square  :"+(x*x));
        }
}
class Sub2 extends MyClass
{
        void calculate(double x)
        {
                System.out.println("Square Root :"+Math.sqrt(x));
        }
}
class Sub3 extends MyClass
{
        void calculate(double x)
        {
                System.out.println("Cube :"+(x*x*x));
        }
}
class AC
{
        public static void main(String arg[])
        {
                Sub1 obj1=new Sub1();
                Sub2 obj2=new Sub2();
                Sub3 obj3=new Sub3();

                obj1.calculate(20);
                obj2.calculate(20);
                obj3.calculate(20);
        }
}
```

Example 2: Java program to illustrate abstract class.

```
// 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();
        }
}
```

Uses of Final:

Final can be used in three ways:

- To prevent modifications to the instance variable
- To Prevent method overriding
- To prevent inheritance

Use 2: To prevent method overriding

```
class Vehicle
{
        final void run()
        {
                System.out.println("running");
        }
}
class Bike extends Vehicle
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}

class FinalMethod
{
        public static void main(String args[])
        {
                Bike b= new Bike();
                b.run();
```

```
        }
}
```

Error Information:

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac FinalMethod.java
FinalMethod.java:10: error: run() in Bike cannot override run() in Vehicle
        void run()
             ^
  overridden method is final
1 error
```

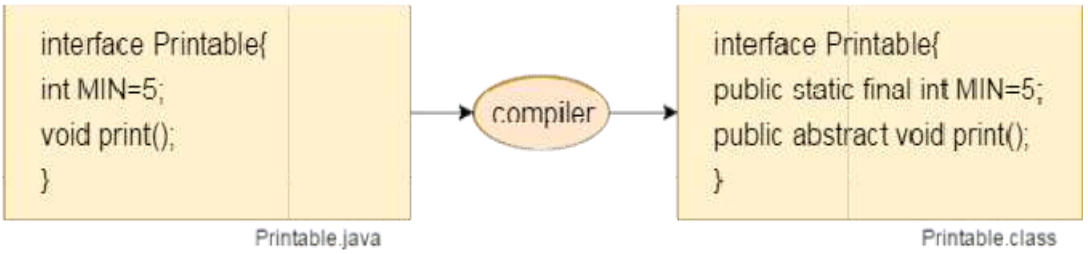
Use 3: To prevent inheritance

```
final class Vehicle
{
        void run()
        {
                System.out.println("running");
        }
}
class Bike extends Vehicle
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}

class FinalClass
{
        public static void main(String args[])
        {
                Bike b= new Bike();
                b.run();
        }
}
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac FinalClass.java
FinalClass.java:8: error: cannot inherit from final Vehicle
class Bike extends Vehicle
                   ^
1 error
```

Interfaces:

- A named collection of method declarations.
- A Java interface is a collection of constants and abstract methods
- Since all methods in an interface are abstract, the abstract modifier is usually left off
- Using interface, you can specify what a class must do, but not how it does.
- Interface fields are public, static and final by default, and methods are public and abstract.



Advantages of interfaces:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritances.

Syntax:

```
access_specifier interface interfae_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;
}
```

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:

```
class classname [extends superclass] [implements interface1 [,interface2...]] {
        // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- The methods that implement an interface must be public. Also, the type signature of implementing method must match exactly the type signature specified in interface definition.

Example 1: Write a java program to implement interface.

```
interface Moveable
{
        int AVG_SPEED=30;
        void Move();
}
class Move implements Moveable
{
         void Move(){
                System .out. println ("Average speed is:  "+AVG_SPEED  );
         }
}
class Vehicle
{
         public static void main (String[] arg)
         {
                Move m = new Move();
                m.Move();
         }
}
```

Example 2: Write a java program to implement interface.

```
 interface Teacher
{
        void display1();
}
interface Student
{
        void display2();
}
class College implements Teacher, Student
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
        public void display2()
        {
                System.out.println("Hi I am Student");
        }
}
class CollegeData
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
        }
}
```

Accessing implementations through interface references:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time,

allowing classes to be created later than the code which calls methods on them.

Ex:

```
interface Test {

   void call();
}

class InterfaceTest implements Test {

   public void call()
   {
      System.out.println("call method called");
   }
}

public class IntefaceReferences {

        public static void main(String[] args)
        {
                Test f ;
                InterfaceTest it= new InterfaceTest();
                f=it;
                f.call();
            }
}
```

Variables in Interfaces:

We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When we include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations.) If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as final variables.

Example: Java program to demonstrate variables in interface.

```
interface left
{
        int i=10;
}
interface right
```

```
{
        int i=100;
}
class Test implements left,right
{
        public static void main(String args[])
        {
                System.out.println(left.i);//10 will be printed
                System.out.println(right.i);//100 will be printed*/
        }
}
```

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.

Example: Java program to demonstrate interfaces can be extended with extend keyword.

```
interface Teacher
{
        void display1();
}
interface Student
{
        void display2();
}
interface T_S extends Teacher, Student
{
        void display3();
}
 class College implements T_S
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
        public void display2()
        {
                System.out.println("Hi I am Student");
        }
        public void display3()
        {
                System.out.println("Hi I am Teacher_Student");
```

```
        }
}
class Class_Interface
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
                c.display3();
         }

}
```

Example 2: Java program to implement interface and inheriting the properties from a class.

```
interface Teacher
{
        void display1();
}
class Student
{
        void display2()
        {
                System.out.println("Hi I am Student");
        }
}
class College extends Student implements Teacher
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
}
class Interface_Class
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
        }
}
```

Difference between Interface and Abstract class:

Abstract Class	Interface
Contains some abstract methods and some concrete methods	Only abstract methods
Contains instance variables	Only static and final variables
Doesn't support multiple inheritance	Supports
public class Apple extends Food { … }	public class Person implements Student,Athlete,Chef { … }

Use of interfaces :

To reveal an object's programming interface (functionality of the object) without revealing its implementation.

- This is the concept of encapsulation.
- The implementation can change without affecting the caller of the interface.
- To have unrelated classes implement similar methods (behaviors).
- One class is not a sub-class of another.
- To model multiple inheritance.
- A class can implement multiple interfaces while it can extend only one class.

Java Nested Interfaces

✓ An interface i.e. declared within another interface or class is known as nested interface.

✓ The nested interfaces are used to group related interfaces so that they can be easy to maintain

✓ The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces :

i) Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

ii) Nested interfaces are declared static implicitly.

Syntax of nested interface which is declared within the interface :

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Syntax of nested interface which is declared within the class :

```
class class_name{
 ...

 interface nested_interface_name{
  ...
 }
}
```

Example using Nested Interfaces (Inside an interface):

```
interface  Showable
 {
  void show();
  interface Message
  {
   void msg();
  }
}
```

```
class TestNestedInterface1  implements  Showable.Message
{
 public void msg()
{
 System.out.println("Hello nested interface");
}

 public static void main(String args[])
{
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Example for nested interface (inside a class) :

```
class A
 {
  interface Message
  {
   void msg();
  }
}

Class NestedIntTest2 implements A.Message
{
 public void msg()
{
System.out.println("Hello nested interface");
}
   public static void main(String args[])
 {
  A.Message message=new NestedIntTest2();   //upcasting here
  message.msg();
 }
}
```

Packages:

- A Package can be defined as a grouping of related types(classes, interfaces)
- A package represents a directory that contains related group of classes and interfaces.
- Packages are used in Java in order to prevent naming conflicts.
- There are two types of packages in Java.
    1. Pre-defined Packages(built-in)
    2. User defined packages

Pre-defined Packages:

Package Name	Description
java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.). This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations. This package is also called as Collections.
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).

java.net	Contains classes for supporting networking operations.
javax.swing	This package helps to develop GUI like java.awt. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.
java.sql	This package helps to connect to databases like Oracle/Sybase/Microsoft Access to perform different operations.

Defining a Package(User defined):

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 pkg;
```

Here, pkg is the name of the package.

For example, the following statement creates a package called MyPackage:

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. 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 a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

Example: Package demonstration

```
package pack;
public class Addition
{
        int x,y;
        public Addition(int a, int b)
        {
                x=a;
                y=b;
        }
        public void sum()
        {
                System.out.println("Sum :"+(x+y));
        }
}
```

Step 1: Save the above file with Addition.java

```
package pack;
public class Subtraction
{
        int x,y;
        public Subtraction(int a, int b)
        {
                x=a;
                y=b;
        }
        public void diff()
        {
                System.out.println("Difference :"+(x-y));
        }
}
```

Step 2: Save the above file with Subtraction.java

Step 3: Compilation

To compile the java files use the following commands

javac -d directory_path name_of_the_java file

Javac –d . name_of_the_java file

Note: -d is a switching options creates a new directory with package name. Directory path represents in which location you want to create package and . (dot) represents

current working directory.

```
D:\Materials\JAVA Material\Unit 2\PackExamples>javac -d . Addition.java

D:\Materials\JAVA Material\Unit 2\PackExamples>javac -d . Subtraction.java
```

Step 4: Access package from another package

There are three ways to use package in another package:

1. With fully qualified name.

```
class UseofPack
{
        public static void main(String arg[])
        {
                pack.Add a=new  pack.Add(10,15);
                a.sum();
                pack.Sub s=new pack.Sub(20,15);
                s.difference();
        }
}
```

2. import package.classname;

```
import  pack.Add;
import  pack.Sub;
 class TestPack
{
        public static void main(String arg[])
        {
                 Add a=new  Add(10,15);
                a.sum();
                Sub  s=new Sub(20,15);
                s.diff();
        }
}
```

3. import package.*;

```
import pack.*;
class UseofPack
{
        public static void main(String arg[])
        {
                 Addition a=new  Addition(10,15);
                a.sum();
                Subtraction s=new Subtraction(20,15);
                s.difference();
        }
}
```

Note: Don't place Addition.java, Subtraction.java files parallel to the pack directory. If you place JVM searches for the class files in the current working directory not in the pack directory.

Access Protection

- Access protection defines actually how much an element (class, method, variable) is exposed to other classes and packages.
- There are four types of access specifiers available in java:
    1. Visible to the class only (private).
    2. Visible to the package (default). No modifiers are needed.
    3. Visible to the package and all subclasses (protected)
    4. Visible to the world (public)

	Private	No Modifier	Protected
Same class	Yes	Yes	Yes
Same package subclass	No	Yes	Yes
Same package non-subclass	No	Yes	Yes
Different package subclass	No	No	Yes

Example:

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. The source for the first package defines three classes: Protection, Derived, and SamePackage.

Name of the package: pkg1

This file is Protection.java

```
package pkg1;

public class Protection
{
        int n = 1;
        private int n_priv = 2;
        protected int n_prot = 3;
        public int n_publ = 4;
```

```
        public Protection()
        {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_priv = " + n_priv);
                System.out.println("n_prot = " + n_prot);
                System.out.println("n_publ = " + n_publ);
        }
}
```

This is file Derived.java:

```
package pkg1;

class Derived extends Protection
{
   Derived()
   {
     System.out.println("Same package - derived (from base) constructor");
     System.out.println("n = " + n);

     /* class only
     *  System.out.println("n_priv = "4 + n_priv); */

     System.out.println("n_prot = " + n_prot);
     System.out.println("n_publ = " +n_publ);
   }
}
```

This is file SamePackage.java

```
package pkg1;

class SamePackage
{
   SamePackage()
   {
     Protection pro = new Protection();
     System.out.println("same package - other constructor");
     System.out.println("n = " + pro.n);

     /* class only
     *  System.out.println("n_priv = " + pro.n_priv); */

     System.out.println("n_prot = " + pro.n_prot);
```

```
            System.out.println("n_publ = " + pro.n_publ);
        }
    }
```

Name of the package: pkg2

This is file Protection2.java:

```
package pkg2;

class Protection2 extends pkg1.Protection
{
        Protection2()
        {
            System.out.println("Other package-Derived (from Package 1-Base) Constructor");

            /* class or package only
            *  System.out.println("n = " + n); */

            /* class only
            *  System.out.println("n_priv = " + n_priv); */

            System.out.println("n_prot = " + n_prot);
            System.out.println("n_publ = " + n_publ);
        }
}
```

This is file OtherPackage.java

```
package pkg2;

class OtherPackage
{
    OtherPackage()
    {
        pkg1.Protection pro = new pkg1.Protection();

        System.out.println("other package - Non sub class constructor");

        /* class or package only
        *  System.out.println("n = " + pro.n); */

        /* class only
        *  System.out.println("n_priv = " + pro.n_priv); */

        /* class, subclass or package only
```

```
        *  System.out.println("n_prot = " + pro.n_prot); */

        System.out.println("n_publ = " + pro.n_publ);
    }
}
```

If you want to try these t two packages, here are two test files you can use. The one for package pkg1 is shown here:

```
/* demo package pkg1 */

package pkg1;

/* instantiate the various classes in pkg1 */
public class Demo
{
    public static void main(String args[])
    {
      Derived obj2 = new Derived();
      SamePackage obj3 = new SamePackage();
    }
}
```

The test file for package pkg2 is

```
package pkg2;

/* instantiate the various classes in pkg2 */
public class Demo2
{
    public static void main(String args[])
    {
      Protection2 obj1 = new Protection2();
      OtherPackage obj2 = new OtherPackage();
    }
}
```

```
D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>javac -d . Demo.java

D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>javac -d . Demo2.java

D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>java pkg1.Demo
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
Same package - derived (from base) constructor
n = 1
n_prot = 3
n_publ = 4
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
same package - other constructor
n = 1
n_prot = 3
n_publ = 4
```

```
D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>java pkg2.Demo2
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
Other package - Derived (from Package 1-Base)Constructor
n_prot = 3
n_publ = 4
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
other package - Non sub class constructor
n_publ = 4
```

Exceptions in Java

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

This is the general form of an exception-handling block:

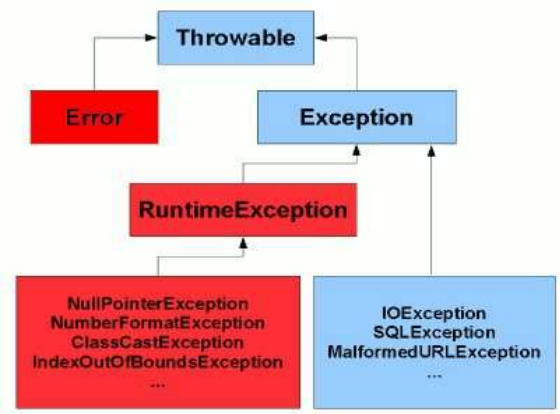
```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed after try block ends
}
```

Here, ExceptionType is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

Exception Types

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.



Uncaught Exception:

This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
        public static void main(String args[]) {
                int d = 0;
                int a = 42 / d;
        }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero at
        Exc0.main(Exc0.java:4)
```

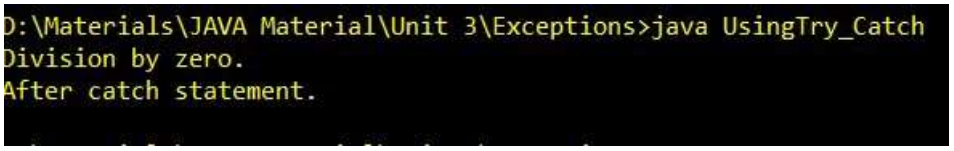
Using try and catch:

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

Example:

```
class UsingTry_Catch
{
        public static void main(String args[])
        {
                int d, a;
                try { // monitor a block of code.
                        d = 0;
                        a = 42 / d;
                        System.out.println("This will not be printed.");
                }
                catch (ArithmeticException e) { // catch divide-by-zero error
                        System.out.println("Division by zero.");
                }
                System.out.println("After catch statement.");
        }
}
```

Output:



Multiple catch Clauses:

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- If one catch statement is executed, the others are bypassed, and execution continues after the try / catch block.
- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes. This is because a catch statement that uses a super class will catch exceptions of that type plus any of its subclasses. Subclass would never be reached if it came after its super class.
- A subclass must come before its super class in a series of catch statements. If not

  unreachable code will be created and a compile time error will result.

Example:

```
// Demonstrate multiple catch statements.
class MultipleCatches
{
        public static void main(String args[])
        {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                }

                catch(ArithmeticException e)
                {
                        System.out.println("Divide by 0: " + e);
                }
                catch(Exception e)
                {
                        System.out.println("Array index out of bounds: " + e);
                }
                System.out.println("After try/catch blocks.");
        }
}
```

Nested try Statements

- The try block within a try block is known as nested try block in java.

Syntax:

```
try{
        try  {
        statement 1;
        statement 2;
        try
         {
           statement 1;
            statement 2;
          }
        catch(Exception e)      {      }
   }

}
catch(Exception e){}
```

- Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular

exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.

- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception (default handler).

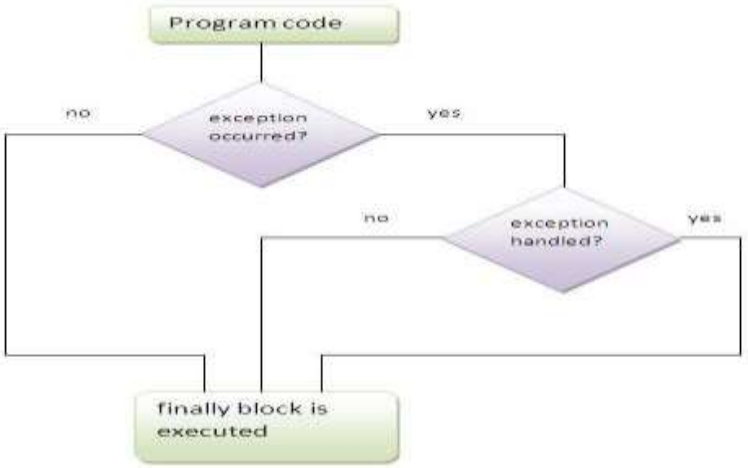
Example:

```
class NestTry
{
    public static void main(String args[])
    {
        try
        {
            int a = args.length;
            /* If no command-line args are present, the following statement will
            generate a divide-by-zero exception. */

            int b = 42 / a;
            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used, then a divide-by-zero
                exception will be generated by the following code. */

                if(a==1)
                    a = a/(a-a); // division by zero

                /* If two command-line args are used, then generate an out-of-
                bounds exception. */
                if(a==2)
                {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Array index out-of-bounds: " + e);
            }
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

```
a = 2
Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException: 42
```

finally block:

- Java finally block is a block that is used to execute important code such as closing connections (databases, network, disks, commit in databases) etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.



Example 1:

```
class Finally_Case1   //exception not occured
{
        public static void main(String args[])
        {
            try{
                    int data=25/25;
                    System.out.println(data);
            }
             finally
             {
                        System.out.println("finally block is always executed");
             }
            System.out.println("rest of the code...");
    }
}
```

Example 2:

```
class Finally_Case2   //exception occured and not handled.
{
        public static void main(String args[])
        {
            try{
                    int data=25/0;
                    System.out.println(data);
            }
            catch(ArrayIndexOutOfBoundsException e)
```

```
                {
                        System.out.println(e);
                    }
                finally
                {
                        System.out.println("finally block is always executed");
                }
                System.out.println("rest of the code...");
        }
    }
```

Example 3:

```
class Finally_Case3   //exception occured and handled.
{
        public static void main(String args[])
        {
            try{
                    int data=25/0;
                    System.out.println(data);
            }
            catch(ArithmeticException e)
            {
                        System.out.println(e);
                    }
            finally
            {
                        System.out.println("finally block is always executed");
            }
            System.out.println("rest of the code...");
        }
    }
```

Use of throw

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

  throw ThrowableInstance;
- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

- There are two ways to obtain a Throwable instance:
  - creating one with the new operator
    throw new exception_class("error message");
  - using the parameter in catch clause - throw exception;

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Syntax:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Example:

```
/*
In this example, we have created the validate method that takes integer value as a parameter.  If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.*/

public class UseofThrow
{
   static void validate(int age)
  {
      try{
          if(age<18)
             throw new ArithmeticException("not valid\n");
          else
                System.out.println("Welcome to participate in voting");
      }
      catch(ArithmeticException e)
      {
          System.out.println(e);
          throw e;
      }
   }
  public static void main(String args[])
  {
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Exceptions and its types:

There are two types of exceptions.

1. Unchecked exceptions
2. Checked exceptions

Unchecked Exceptions: Found during running of a program

ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Checked Exceptions: Found at compilation time

ClassNotFoundException	Class not found.
IllegalAccessException	Access to a class is denied.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating user-defined exception:

- We can also create our own exception by creating a sub class simply by extending java Exception class.
- Define a constructor for Exception sub class (not compulsory) and override the toString() method to display customized message in catch clause.
  - Exception();
  - Exception(parameter);
- The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Syntax of toString():

- String toString( )

- Returns a String object containing a description of the exception.
- This method is called by println( ) when outputting a Throwable object.
- belongs to Object class

Example:

```
class MyException extends Exception
{
        String s ;
        MyException( String s)
        {
                this.s=s;
        }
        public String toString()
        {
            return ("User Defined " +s) ;
        }
}
class UserDefinedException
{
        public static void main(String args[])
        {
          try{
                        throw new MyException("Exeption"); // throw is used to create
                        //a new exception and throw it.
          }
```

```
            catch(MyException e)
            {
                    System.out.println(e) ;
            }
        }
    }
```

Threading in JAVA

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.

- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

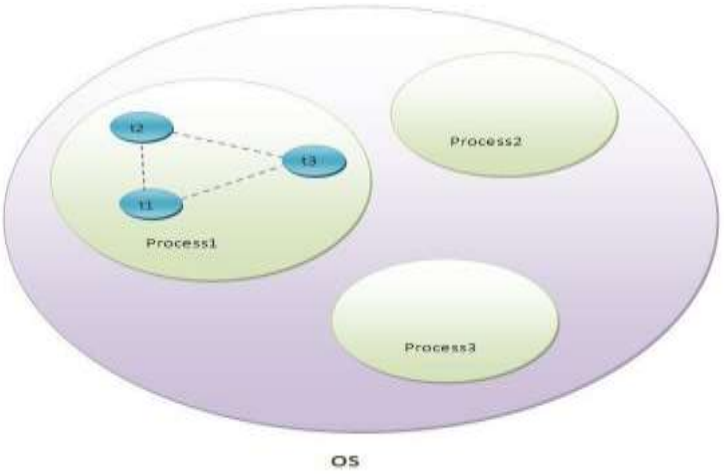
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

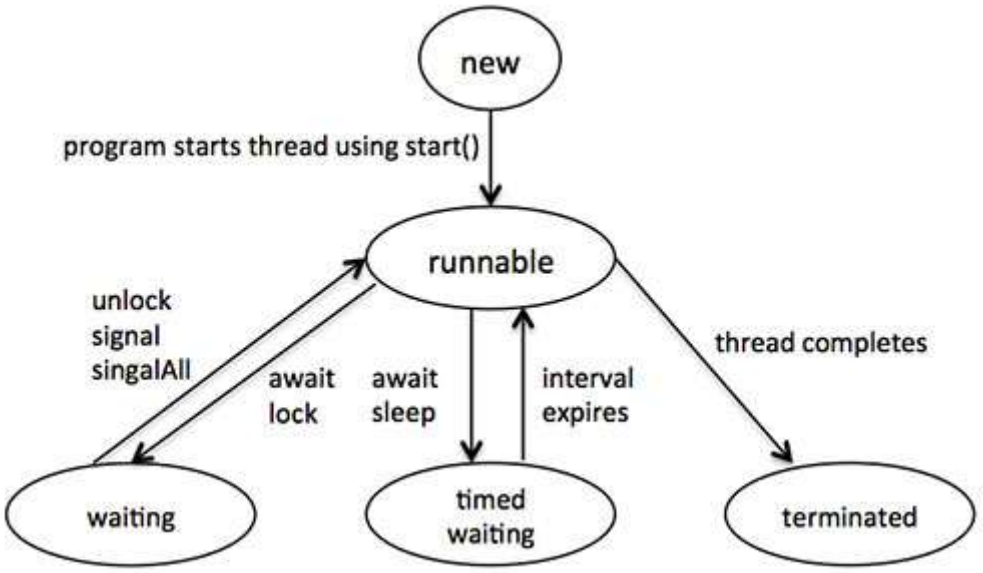
Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Life Cycle of a Thread:

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



Above-mentioned stages are explained here:

New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting: Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Ways of Creating Threads

I) Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing Runnable interface. You will need to follow three basic steps:

Step 1: As a first step you need to implement a run() method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

Step 2: At second step you will instantiate a Thread object using the following constructor: Thread(Runnable threadObj, String threadName);

Where, threadObj is an instance of a class that implements the Runnable interface andthreadName is the name given to the new thread.

Step 3: Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

```
void start( );
```

NOTE: Follow Example Program in notes

II) **Create Thread by Extending Thread Class**:

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override run( ) method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

```
void start( );
```

NOTE: Follow Example Program in notes

Thread class Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

SN	Methods with Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack()

	Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

isAlive () and join() Methods :

```
final boolean isAlive()
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

```
final void join() throws InterruptedException

final void join(long milliseconds) throws
InterruptedException
```

This method waits until the thread on which it is called terminates.

Using join(), we tell thread to wait until the specified thread completes its execution.

There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

Example for isAlive() :

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("r1 ");
        try {
            Thread.sleep(500);
        }
    catch(InterruptedException ie)
```

```
        {
                System.out.println("Interrupted");

        }
        System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
            MyThread t1=new MyThread();
            MyThread t2=new MyThread();
            t1.start();
            t2.start();
            System.out.println(t1.isAlive());
            System.out.println(t2.isAlive());
        }
}
```

Example Program using join():-

```
public class MyThreadJTest extends Thread
{

public void run()
        {
        System.out.println("r1 ");
        try {
            Thread.sleep(500);
            }
      catch(InterruptedException ie)
        {
```

```
                System.out.println("Interrupted");
            }
        System.out.println("r2 ");
        }

    public static void main(String[] args)
        {
            MyThreadJTest t1=new MyThreadJTest();
            MyThreadJTest t2=new MyThreadJTest();
            t1.start();

            try
              {
                  t1.join();  //Waiting for t1 to finish
            }catch(InterruptedException ie)
            {System.out.println(" main thread Interrupted");
             }

            t2.start();
        }
    }
```

Creation of Multiple Threads :

The following example demonstrates how to create multiple threads.

Example program for multiple thread creation :

```
        class NewThread implements Runnable
  {
 String name;
 Thread t;
 NewThread(String n)
 {
  name=n;
  t=new Thread(this.name);
  System.out.println("Child Thread:"+t);
  t.start();
 }
 public void run()
 {
  try
  {
   for(int i=5;i>0;i--)
   {
    System.out.println(name+":"+i);
     Thread.sleep(500);
   }
   }
   catch(InterruptedException e)
   {
      System.out.println(name+"Interrupted");
   }
    System.out.println(name+"Terminated!!");
  }
}
class ThreadDemo
{
 public static void main(String args[])
 {
  new NewThread("One");
```

```
    new NewThread("Two");
    new NewThread("Three");
    try
    {

        Thread.sleep(1000);
    }
    catch(InterruptedException e)
    {
      System.out.println("Main Thread Interrupted");
    }
    System.out.println("Main Thread Terminated");
   }
}
```

Java synchronization:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Synchronization in Java:

Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization: The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.

    NOTE: Follow Example Program Given in Notes and give explanation for it

2. Cooperation (Inter-thread communication in java)

Inter-thread communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

1) **wait() method**

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

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

2**) notify() method**

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

3) **notifyAll() method**

Wakes up all threads that are waiting on this object's monitor.

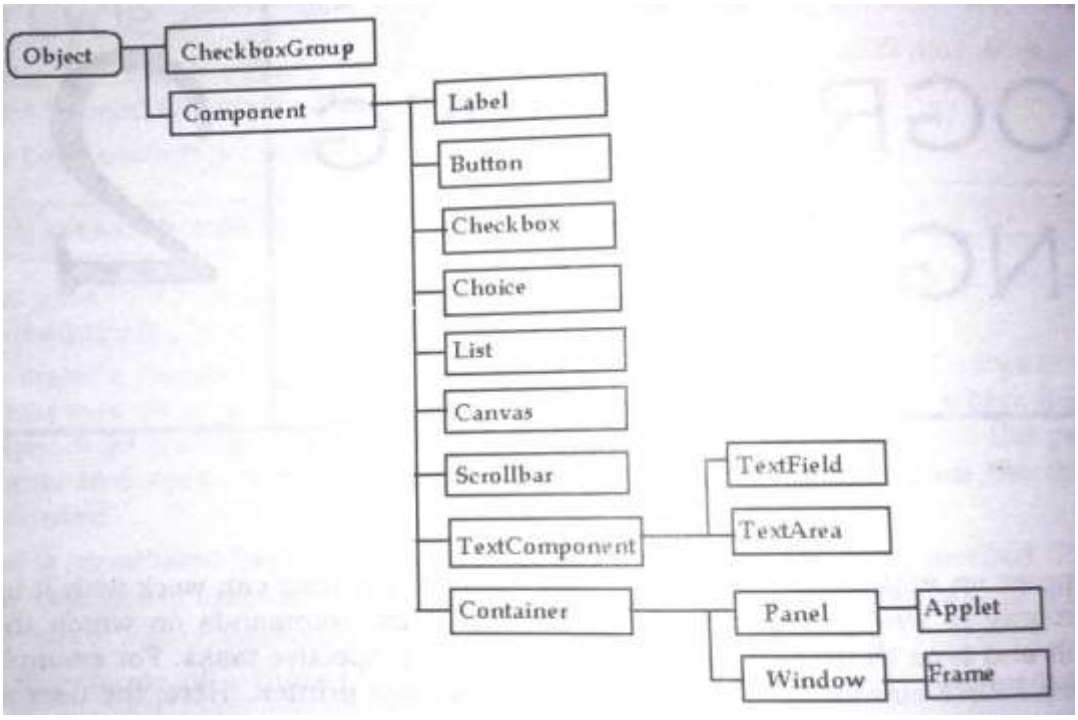
Syntax:

public final void notifyAll()

Example : Follow Exercise 7 c and it's additional Program

AWT (Abstract Window Toolkit):

AWT represents a class library to develop applications using GUI. The **java.awt** package consists of classes and interfaces to develop GUIs.

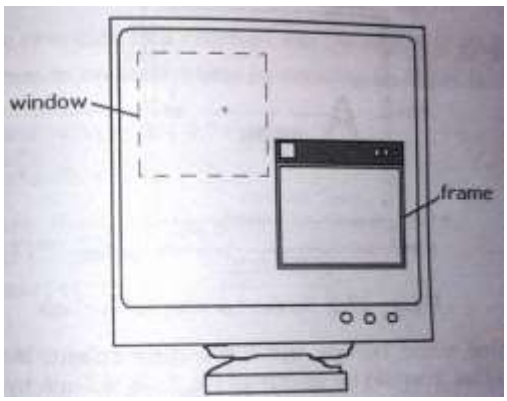


Component: A component represents an object which is displayed pictorially on the screen and interacts with the user.

Ex. Button, TextField, TextArea

Container: A Container is a subclass of Component; it has methods that allow other components to be nested in it. A container is responsible for laying out (that is positioning) any component that it contains. It does this with various layout managers.

Panel: Panel class is a subclass of Container and is a super class of Applet. When screen output is redirected to an applet, it is drawn on the surface of the Panel object. In, essence panel is a window that does not contain a title bar, menu bar or border.

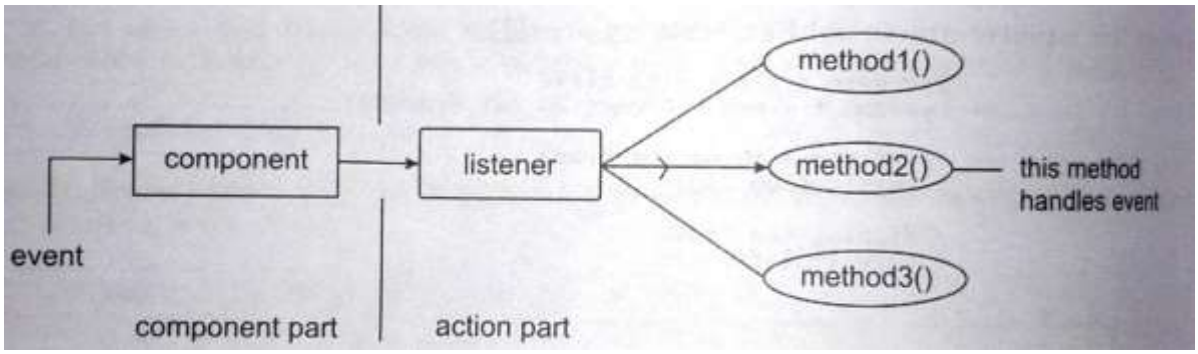


Window: A window represents a rectangular area on the screen without any borders or title bar. The Window class create a top-level window.

Frame: It is a subclass of Window and it has title bar, menu bar, border and resizing windows.

Delegation Event Model:

The modern approach (from version 1.1 onwards) to handle events is based on the delegation event model. Its concept is quite simple: a source generates an event and sends it to one or more listeners.



In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

Events: An *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a GUI. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources: A source is an object that generates an event. Generally sources are components. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener (TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ).

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Event Listeners: A listener is an object that is notified when an event occurs. It has two major requirements.

1.It must have been registered with one or more sources to receive notifications aboutspecific types of events.
2. It must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in *java.awt.event* package.

Sources of Events:

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked;
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Classes and Listener Interfaces:

The java.awt.event package provides many event classes and Listener interfaces for event handling. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the super class for all events. Its one constructor is shown here:

EventObject(Object *src*) - Here, *src* is the object that generates this event.

EventObject contains two methods:

getSource( ) - returns the source of the event.
toString( ) - toString( ) returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

int getID( )

.

The package **java.awt.event** defines many types of events that are generated by various user interface elements

Event Class	Description	Listener Interface
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.	ActionListener
AdjustmentEvent	Generated when a scroll bar is manipulated.	AdjustmentListener
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.	ComponentListener
ContainerEvent	Generated when a component is added to or removed from a container.	ContainerListener
FocusEvent	Generated when a component gains or losses keyboard focus.	FocusListener
InputEvent	Abstract super class for all component input event classes.	
ItemEvent	Generated when a check box or list item is clicked	ItemListener
KeyEvent	Generated when input is received from the keyboard.	KeyListener
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.	MouseListener and MouseMotionListener
TextEvent	Generated when the value of a text area or text field is changed.	TextListener
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.	WindowListener

Useful Methods of Component class:

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

The ActionEvent Class:

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK (Ex. Escape) , and SHIFT_MASK.

ActionEvent has these three constructors:

- ActionEvent(Object src, int type, String cmd)
- ActionEvent(Object src, int type, String cmd, int modifiers)
- ActionEvent(Object src, int type, String cmd, long when, int modifiers)

You can obtain the command name for the invoking ActionEvent object by using the getActionCommand( ) method, shown here:

String getActionCommand( )

The AdjustmentEvent Class:

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events.

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

The ComponentEvent Class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.

The **getComponent( )** method returns the component that generated the event. It is shown here:

Component getComponent( )

The ContainerEvent Class:

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**.

The FocusEvent Class:

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

The InputEvent Class:

The abstract class **InputEvent** is a subclass of **ComponentEven**t and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the InputEvent class defined the following eight values to represent the modifiers:

ALT_MASK	**ALT_GRAPH_MASK**	**BUTTON2_MASK**	**BUTTON3_MASK**
BUTTON1_MASK	**CTRL_MASK**	**META_MASK**	**SHIFT_MASK**

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	**ALT_GRAPH_DOWN_MASK**	**BUTTON1_DOWN_MASK**
BUTTON2_DOWN_MASK	**BUTTON3_DOWN_MASK**	**CTRL_DOWN_MASK**
META_DOWN_MASK	**SHIFT_DOWN_MASK**	

The KeyEvent Class

A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED, KEY_RELEASED,** and **KEY_TYPED.**

The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing shift does not generate a character.

There are many other integer constants that are defined by KeyEvent. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters.

The MouseEvent Class:

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse
MOUSE_DRAGGED	The user dragged the mouse
MOUSE_ENTERED	The mouse entered a component
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```
int getX( )
int getY( )
```

The TextEvent Class:

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant **TEXT_VALUE_CHANGED.**

The WindowEvent Class:

The **WindowEvent** class defines integer constants that can be used to identify different types of events:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window was iconified.
WINDOW_ICONIFIED	The window gained input focus.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.

EventListener Interfaces:

An event listener registers with an event source to receive notifications about the events of a particular type. Various event listener interfaces defined in the `java.awt.event` package are given below:

Interface	Description
ActionListener	Defines the actionPerformed() method to receive and process action events. *void actionPerformed(ActionEvent ae)*
MouseListener	Defines five methods to receive mouse events, such as when a mouse is clicked, pressed, released, enters, or exits a component *void mouseClicked(MouseEvent me)* *void mouseEntered(MouseEvent me)* *void mouseExited(MouseEvent me)* *void mousePressed(MouseEvent me)* *void mouseReleased(MouseEvent me)*
MouseMotionListener	Defines two methods to receive events, such as when a mouse is dragged or moved. *void mouseDragged(MouseEvent me)* *void mouseMoved(MouseEvent me)*
AdjustmentListner	Defines the adjustmentValueChanged() method to receive and process the adjustment events. *void adjustmentValueChanged(AdjustmentEvent ae)*
TextListener	Defines the textValueChanged() method to receive and process an event when the text value changes. *void textValueChanged(TextEvent te)*
WindowListener	Defines seven window methods to receive events. *void windowActivated(WindowEvent we)* *void windowClosed(WindowEvent we)* *void windowClosing(WindowEvent we)* *void windowDeactivated(WindowEvent we)* *void windowDeiconified(WindowEvent we)* *void windowIconified(WindowEvent we)* *void windowOpened(WindowEvent we)*
ItemListener	Defines the itemStateChanged() method when an item has been *void itemStateChanged(ItemEvent ie)*
WindowFocusListener	This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here: *void windowGainedFocus(WindowEvent we)* *void windowLostFocus(WindowEvent we)*
ComponentListener	This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here: *void componentResized(ComponentEvent ce)* *void componentMoved(ComponentEvent ce)* *void componentShown(ComponentEvent ce)* *void componentHidden(ComponentEvent ce)*

ContainerListener	This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here: *void componentAdded(ContainerEvent ce)* *void componentRemoved(ContainerEvent ce)*
FocusListener	This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here: *void focusGained(FocusEvent fe)* *void focusLost(FocusEvent fe)*
KeyListener	This interface defines three methods. *void keyPressed(KeyEvent ke)* *void keyReleased(KeyEvent ke)* *void keyTyped(KeyEvent ke)*

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener
2. Implement the concerned interface

Registration Methods:

For registering the component with the Listener, many classes provide the registration methods. For example:

Button
- public void addActionListener(ActionListener a){}

MenuItem
- public void addActionListener(ActionListener a){}

TextField
- public void addActionListener(ActionListener a){}
- public void addTextListener(TextListener a){}

TextArea
- public void addTextListener(TextListener a){}

Checkbox
- public void addItemListener(ItemListener a){}

Choice
- public void addItemListener(ItemListener a){}

List
- public void addActionListener(ActionListener a){}
- public void addItemListener(ItemListener a){}

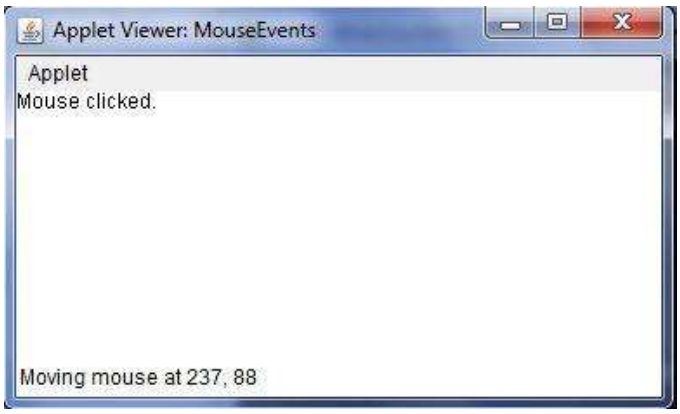
Mouse
- public void addMouseListener(MouseListener a){}

Handling Mouse Events Example Program:

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*; /*

<applet code="MouseEvents" width=300
height=100> </applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
{
        String msg = "";
        int mouseX = 0, mouseY = 0; // coordinates of mouse
        public void init()
        {
                addMouseListener(this);
                addMouseMotionListener(this);
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {
                // save coordinates
                mouseX = 0;
                mouseY = 10;
                msg = "Mouse
                clicked."; repaint();
        }
        // Handle mouse entered.
        public void mouseEntered(MouseEvent me)
        {
                // save coordinates
                mouseX = 0;
                mouseY = 10;
                msg = "Mouse entered.";
                repaint();
        }
        // Handle mouse exited.
        public void mouseExited(MouseEvent me)
        {
                // save coordinates
                mouseX = 0;
                mouseY = 10;
                msg = "Mouse exited.";
                repaint();
        }
        // Handle button pressed.
        public void mousePressed(MouseEvent me)
        {
                // save coordinates
                mouseX = me.getX();
```

```
            mouseY = me.getY();
            msg = "Down";
            repaint();
        }
        // Handle button released.
        public void mouseReleased(MouseEvent me)
        {
            // save coordinates
            mouseX =
            me.getX(); mouseY
            = me.getY(); msg =
            "Up"; repaint();
        }
        // Handle mouse dragged.
        public void mouseDragged(MouseEvent me)
        {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "*";
            showStatus("Dragging mouse at " + mouseX + ", " +
            mouseY); repaint();
        }
        // Handle mouse moved.
        public void mouseMoved(MouseEvent me)
        {
            // show status
            showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
        }
        // Display msg in applet window at current X,Y
        location. public void paint(Graphics g)
        {
            g.drawString(msg, mouseX, mouseY);
        }
}
```

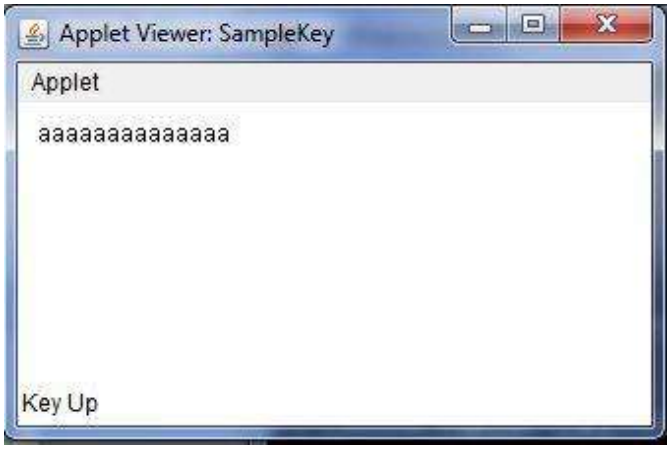
Output:



Handling Key Board Events:

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*; /*

<applet code="SampleKey" width=300 height=100>
</applet>
*/
public class SampleKey extends Applet implements KeyListener
{
        String msg = "";

        public void init() {
                addKeyListener(this);
        }
        public void keyPressed(KeyEvent ke) {
                showStatus("Key Down");
        }
        public void keyReleased(KeyEvent ke) {
                showStatus("Key Up");
        }
        public void keyTyped(KeyEvent ke) {
                msg += ke.getKeyChar();
                repaint();
        }
        // Display keystrokes.
        public void paint(Graphics g) {
                g.drawString(msg, 10, 20);
        }
}
```

Output:



Handling Action Event Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="ButtonEvent3" width=300 height=100>
</applet>
*/
public class ButtonEvent3 extends Applet implements ActionListener
 {
        Button a ;
        String msg;
        public void init()
        {
                a=new Button("PVPSIT");

                add(a);

                a.addActionListener(this);
         }
        public void actionPerformed(ActionEvent ae)
        {
                String str=ae.getActionCommand();
                if(str.equals("PVPSIT"))
                msg="You pressed PVPSIT";
                repaint();

        }
        public void paint(Graphics g)
        {
                g.drawString(msg,100,100);

        }
 }
```

Output:

Adapter Classes:

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

For example,

MouseListener	MouseAdapter
void mouseClicked(MouseEvent me)	*void mouseClicked(MouseEvent me){ }*
void mouseEntered(MouseEvent me)	*void mouseEntered(MouseEvent me) { }*
void mouseExited(MouseEvent me)	*void mouseExited(MouseEvent me) { }*
void mousePressed(MouseEvent me)	*void mousePressed(MouseEvent me) { }*
void mouseReleased(MouseEvent me)	*void mouseReleased(MouseEvent me) { }*

Table: Commonly used Listener Interfaces implemented by Adapter Classes

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
  public void init() {
        addMouseListener(new MyMouseAdapter(this));

  }
}
class MyMouseAdapter extends MouseAdapter
{
  AdapterDemo ad;
  public MyMouseAdapter(AdapterDemo ad)
  {
        this.ad = ad;
```

```
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me)
    {
            ad.showStatus("Mouse clicked");
    }
}
```

Inner Classes:

I*nner class* is a class defined within another class, or even within an expression.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="InnerClassDemo" width=300 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
        String msg = "hello";

        public void init() {
                addKeyListener(new MyKeyIn());
        }
        class MyKeyIn extends KeyAdapter
        {
                public void keyPressed(KeyEvent ke) {
                        showStatus("Key Pressed");
                }
        }

        public void paint(Graphics g) {
                g.drawString(msg, 10, 20);
        }
}
```

Anonymous Inner Classes:

An *anonymous* inner class is one that is not assigned a name.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AInnerClassDemo" width=300 height=100>
</applet> */
```

```
public class AInnerClassDemo extends Applet
{
  String msg = "hello";

  public void init()
  {
        addKeyListener(new KeyAdapter(){
              public void keyPressed(KeyEvent ke) {
              showStatus("Key Pressed");
             }
        });
  }
  // Display keystrokes.
  public void paint(Graphics g) {
        g.drawString(msg, 10, 20);
  }
}
```

Control Fundamentals:

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

These controls are subclasses of **Component**

Adding and Removing Controls: To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add()**, which is defined by **Container**. The General form is:

Component add(Component *compObj*)

Here, *compObj* is an instance of the control that you want to add. A reference to *compObj* is returned.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. Here is one of its forms:

void remove(Component *obj*)

Here, *obj* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

The HeadlessException:

Most of the AWT controls have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present).

Labels:

A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label( ) throws HeadlessException Label(String *str*)
throws HeadlessException Label(String *str*, int *how*)
throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

Using Buttons:

A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)
String getLabel( )

Here, *str* becomes the new label for the button

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="ButtonEvent1" width=300 height=100>
</applet>
*/
public class ButtonEvent1 extends Applet
 {
   Button b,b1;
   public void init()
   {
        b=new Button("PVPSIT");
        b1=new Button();
        add(b);
        add(b1);
   } }
```

Check Boxes:

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors: Checkbox( ) throws
HeadlessException Checkbox(String *str*) throws
HeadlessException Checkbox(String *str*, boolean *on*)
throws HeadlessException
Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) throws HeadlessException
Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. The value of *on* determines the initial state of the check box.

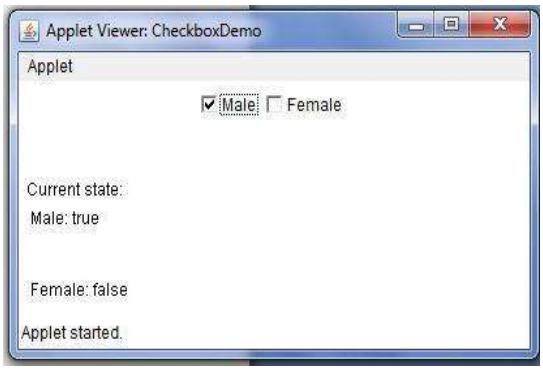
Methods:

boolean getState( ) - To retrieve the current state of a check box
void setState(boolean *on*) - to set the state of a check box String
getLabel( ) – returns the label associated with check box void
setLabel(String *str*) – to set the label

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener
{
        String msg = "";
        Checkbox m,f;
        public void init()
        {
                m = new Checkbox("Male", true);
                f = new Checkbox("Female");

                add(m );
                add(f);
```

```
            m.addItemListener(this);
            f.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
            repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g) {
            msg = "Current state: ";
            g.drawString(msg, 6, 80); msg
            = " Male: " + m.getState();
            g.drawString(msg, 6, 100);

            msg = " Female: " + f.getState();
            g.drawString(msg, 6, 150);
        }
}
```



CheckboxGroup:

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons* —only one button can be selected at any one time.

To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**.

Only the default constructor is defined, which creates an empty group.

Methods:

Checkbox getSelectedCheckbox( ) - which check box in a group is currently selected void setSelectedCheckbox(Checkbox *which*) - *which* is the check box that you want to be selected. The previously selected check box will be turned off

Example:

```
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=240 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener
{
    String msg = "";
    Checkbox m,f;
```

```
CheckboxGroup cbg;
public void init()
{
        cbg = new CheckboxGroup();
        m = new Checkbox("Male", cbg, true); f
        = new Checkbox("Female", cbg, false);

        add(m);
        add(f);

        m.addItemListener(this);
        f.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
        repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g)
{
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
}
}
```

Choice Controls:

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. **Choice** defines only the default constructor, which creates an empty list. To add a selection to the list, call **add( )**. It has this general form:

void add(String *name*) - *name* is the name of the item being added.

Items are added to the list in the order in which calls to **add( )** occur.

Methods:

String getSelectedItem( ) – returns the item which is currently selected
int getSelectedIndex( ) - returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.
int getItemCount( ) – returns number of items in the list
void select(int *index*) - to set the currently selected item with index void
select(String *name*) - to set the currently selected item with a string
String getItem(int *index*) – returns the name associated with the index

Example:

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

```
import java.applet.*;
/*
        <applet code="ChoiceDemo" width=300
        height=180> </applet>
*/
public class ChoiceDemo extends Applet implements ItemListener
{
        Choice college ;
        String msg = "";
        public void init()
        {
                college = new Choice();

                // add items to os list
                college.add("PVPSIT");
                college.add("BEC");
                college.add("RVR&JC");
                college.add("VRSEC");

                add(college);

                // register to receive item events
                college.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
        // Display current selections.
        public void paint(Graphics g)
        {
                msg = "Selected College is: ";
                msg += college.getSelectedItem();
                g.drawString(msg, 6, 120);
        }
}
```

List:

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections.

List provides these constructors:

List( ) throws HeadlessException

List(int *numRows*) throws HeadlessException

List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms:

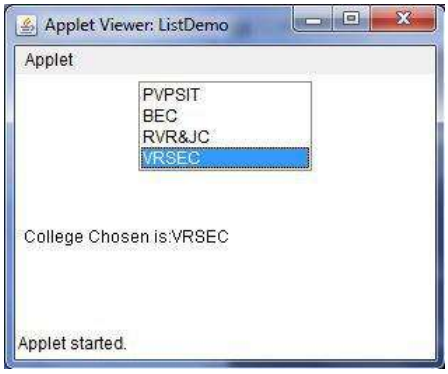
void add(String *name*)

void add(String *name*, int *index*)

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
        <applet code="ListDemo" width=300 height=180>
        </applet>
*/
public class ListDemo extends Applet implements ActionListener
{
        List college;
        String msg = "";
        public void init()
        {
                college = new List(4,true);

                college.add("PVPSIT");
                college.add("BEC");
                college.add("RVR&JC");
                college.add("VRSEC");

                //college.select(1);
                add(college);

                // register to receive action events
                college.addActionListener(this);
                }
        public void actionPerformed(ActionEvent ae) {
                repaint();
        }
```



```
        // Display current selections.
        public void paint(Graphics g)
        {
                msg="College Chosen is:";
                int ind[];

                ind = college.getSelectedIndexes();
                for(int i=0; i<ind.length; i++)
                        msg += college.getItem(ind[i]) + " ";
                g.drawString(msg, 6, 120);
        }
}
```

TextField:

The **TextField** class implements a single-line text-entry area. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.

TextField is a subclass of **TextComponent**. **TextField** defines the following constructors:

TextField( ) throws HeadlessException TextField(int *numChars*) throws HeadlessException TextField(String *str*) throws HeadlessException TextField(String *str*, int *numChars*) throws HeadlessException

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.

Methods:

String getText( ) - To obtain the string currently contained in the text field void setText(String *str*) - To set the text, here, *str* is the new string.

String getSelectedText( ) - returns currently selected text

void select(int *startIndex*, int *endIndex*) - selects the characters beginning at *startIndex* and ending at *endIndex* –1.

boolean isEditable( ) – returns boolean value (true/false)

void setEditable(boolean *canEdit*) - if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

void setEchoChar(char *ch*) – specified echo character will be displayed in TextField

boolean echoCharIsSet( ) –returns true or false

char getEchoChar( ) – returns the echo character

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet implements TextListener
{
        TextField name, pass;
        public void init()
        {
                Label namep = new Label("Name: " );
                name = new TextField(12);

                Label passp = new Label("Password: " );
                pass = new TextField(8);
                pass.setEchoChar('*');

                add(namep);
                add(name);
                add(passp);
                add(pass);
                // register to receive action events
                name.addTextListener(this);
                pass.addTextListener(this);
        }
        // User pressed Enter.
        public void textValueChanged(TextEvent ae)
        {
                repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString("Name: " + name.getText(), 6, 60);
                g.drawString("Password: " + pass.getText(), 6, 100);
                g.drawString("Selected text in name: "+ name.getSelectedText(), 6, 80);
        }
}
```

TextArea:

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH
SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY
SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described in the preceding section.

TextArea adds the following methods:

void append(String *str*) - appends the string specified by *str* to the end of the current void insert(String *str*, int *index*) - inserts the string passed in *str* at the specified index void replaceRange(String *str*, int *startIndex*, int *endIndex*) - replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet
{
    public void init()
    {
        String val = "Java 7 is the latest version of the most widely-used computer
                language for Internet programming.";
        TextArea text = new TextArea(val, 10, 30);
        add(text);
    }
}
```

Managing Scroll Bars:

Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

Scroll bars are encapsulated by the **Scrollbar** class. **Scrollbar** defines the following constructors:

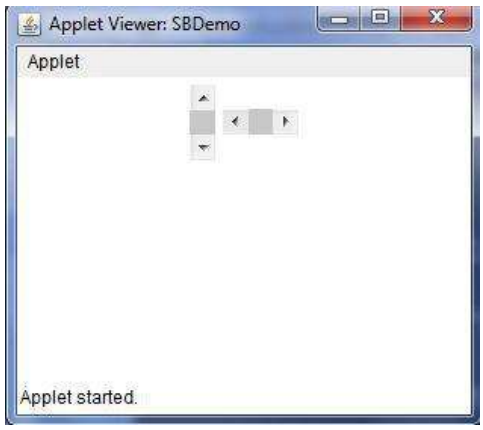
Scrollbar( ) throws HeadlessException
Scrollbar(int *style*) throws HeadlessException
Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*) throws HeadlessException

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

Methods:

void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)	If we construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues()**
int getValue( )	To get the current value
void setValue(int *newValue*)	TO set the current value
int getMinimum( )	To get the minimum value
int getMaximum( )	To get the maximum value

Example:




```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
        <applet code="SBDemo" width=300 height=200>
        </applet>
*/
public class SBDemo extends Applet
{
         Scrollbar vertSB, horzSB;

        public void init()
        {
                vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 100);
                horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 100);
                add(vertSB);
                add(horzSB);
        }
}
```

Layout Manager

A layout manager is a class that is useful to arrange components in a particular manner in container or a frame.

Java soft people have created a LayoutManager interface in java.awt package which is implemented in various classes which provide various types of layouts to arrange the components. The following classes represents the layout managers in Java:

1. FlowLayout
2. BorderLayout
3. GridLayout
4. CardLayout
5. GridBagLayout
6. BoxLayout

To set a particular layout, we should first create an object to the layout class and pass the object to setLayout() method. For example, to set FlowLayout to the container:

FlowLayout obj=new FlowLayout();

c. setLayout(obj); // assume c is container

FlowLayout:

FlowLayout is useful to arrange the components in a line one after the other. When a line is filled with components, they are automatically placed in a next line. This is the default layout in applets.

Constructors:

FlowLayout( )

FlowLayout(int how)

FlowLayout(int how, int horz, int vert)

The first form creates the default layout, which centres components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for how are as follows:

FlowLayout.LEFT

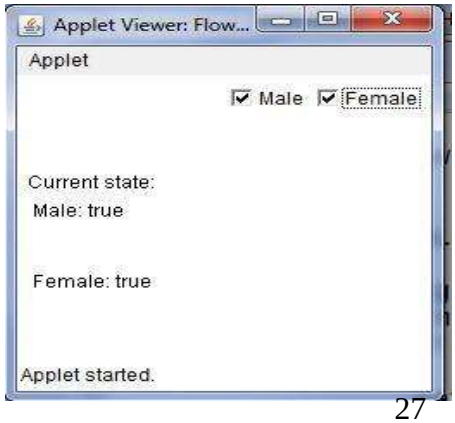
FlowLayout.CENTER

FlowLayout.RIGHT

The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=240
height=200>
</applet>
*/
```



```
public class FlowLayoutDemo extends Applet implements ItemListener
{
	String msg="";
	Checkbox m,f;
	public void init()
	{
		setLayout(new FlowLayout(FlowLayout.RIGHT));
		m = new Checkbox("Male", true); f = new
		Checkbox("Female");
		add(m );
		add(f);
		m.addItemListener(this);
		f.addItemListener(this);
	}
	public void itemStateChanged(ItemEvent ie)
	{
		repaint();
	}
	// Display current state of the check boxes.
	public void paint(Graphics g) {
		msg = "Current state: ";
		g.drawString(msg, 6, 80); msg
		= " Male: " + m.getState();
		g.drawString(msg, 6, 100);

		msg = " Female: " + f.getState();
		g.drawString(msg, 6, 150);
	}
}
```

(or)

```
/*
<applet code="FlowLayoutDemo" width=240 height=200>
</applet> */
public class FlowLayoutDemo extends Applet
{
	Checkbox m,f;
	public void init()
	{
		setLayout(new
FlowLayout(FlowLayout.RIGHT));
		m = new Checkbox("Male", true);
		f = new Checkbox("Female");

		add(m );
		add(f);


	}
}
```

BorderLayout:

BorderLayout is useful to arrange the components in the four borders of the frame as well as in the centre. The borders are identified with the names of the directions. The top border is specified as 'North', the right side border as 'East', the bottom one as 'South' and the left one as 'West'. The centre is represented as 'Centre'.

Constructors:

BorderLayout( )

BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.NORTH

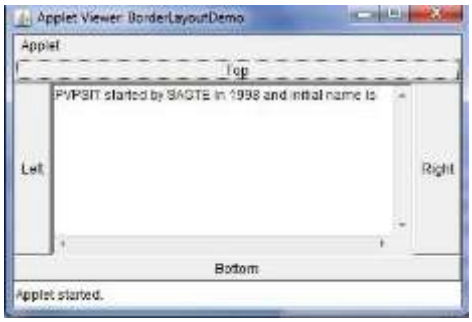
When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

void add(Component *compObj*, Object *region*)

Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

Example:

```
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo 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 = "PVPSIT started by SAGTE in 1998.\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```



GridLayout:

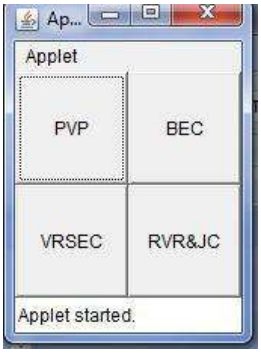
GridLayout is useful to divide the container into a 2D grid form that contains several rows and columns. The container is divided into equal-sized rectangle; and one component is placed in each rectangle.

Constructors:

GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimitedlength columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Example:

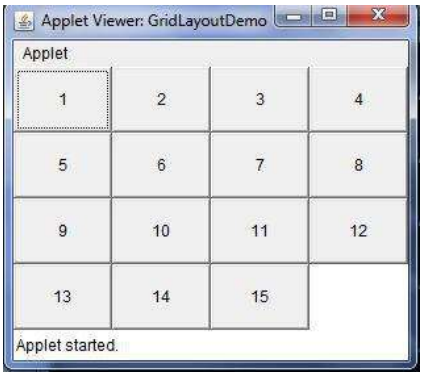



```
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo2" width=150 height=150>
</applet>
*/
public class GridLayoutDemo2 extends Applet
{
	Button b1,b2,b3,b4;
	public void init()
	{
		setLayout(new GridLayout(2, 2));
		b1=new Button("PVP");
		b2=new Button("BEC");
		b3=new Button("VRSEC");
		b4=new Button("RVR&JC");

		add(b1);
		add(b2);
		add(b3);
		add(b4);
	}
}
```

or

```
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
```

```
</applet>
*/
public class GridLayoutDemo extends Applet
{
        static final int n = 4;
        public void init() {
                setLayout(new GridLayout(n, n));

                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));
                        }
                }
        }
}
```



CardLayout:

A CardLayout object is a layout manager which treats each component as a card. Only one card is displayed at a time, and the container acts as a stack of cards. The first component added to a CardLayout object is visible component when the container is first displayed.

CardLayout provides these two constructors:

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

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. Finally, you add this pane to the window.

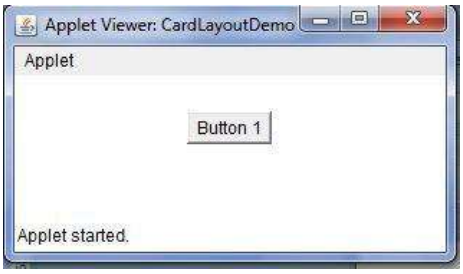
Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

void add(Component *panelObj*, Object *name*)
or
void add(Object name, Component *panelObj*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelObj*. After you have created a deck, your 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*)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet implements ActionListener
{
        Button b1,b2,b3,b4;
        Panel p;
        CardLayout card;
        public void init()
        {
                b1 = new Button("Button 1");
                b2 = new Button("Button 2");
                b3 = new Button("Button 3");
                b4 = new Button("Button 4");

                p=new Panel();
                card=new CardLayout(20,20);
                p.setLayout(card);

                p.add("First",b1);
                p.add("Second",b2);
                p.add("Third",b3);
                p.add("Fourth",b4);

                b1.addActionListener(this);
                b2.addActionListener(this);
                b3.addActionListener(this);
                b4.addActionListener(this);
                add(p);
        }
        public void actionPerformed(ActionEvent ae)
        {
                card.next(p);
        }
}
```



GridBagLayout:

A GridBagLayout class represents grid bag layout manager where the components are arranged in rows and columns. In this layout the component can span more than one row or column and the size of the component can be adjusted to fit the display area.

When positioning the components by using grid bag layout, it is necessary to apply some constraints or conditions on the components regarding their position, size and place in or around the components etc. Such constraints are specified using GridBagConstrinats class.

In order to create GridBagLayout, we first instantiate the GridBagLayout class by using its only no-argument constructor

```
GridBagLayout layout=new GridBagLayout();
setLayout(layout);
```

and defining it as the current layout manager.

To apply constraints on the components, we should first create an object to GridBagConstrinats class, as

```
GridBagConstrinats gbc =new GridBagConstrinats();
```

This will create constraints for the components with default value. The other way to specify the constraints is by directly passing their values while creating the GridBagConstrinats as

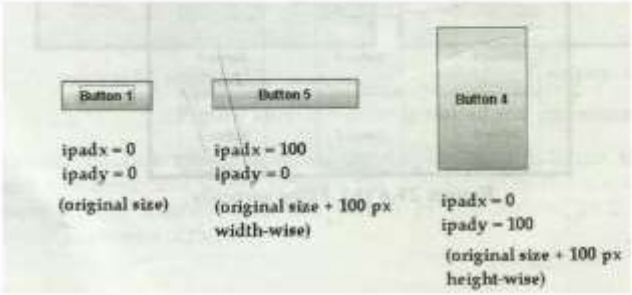
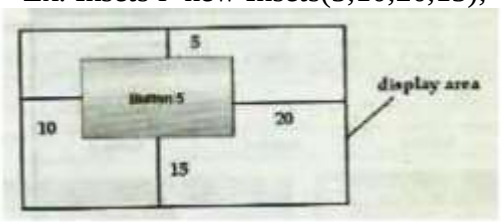
```
GridBagConstrinats gbc= new GridBagConstrinats(
    int gridx, int gridy, int gridwidth, int gridheight, double weightx, double
    weighty, int anchor, int fill, Insets insets, int ipadx, int ipady );
```

To set the constraints use setConstraints() method in GridBagConstrinats class and its prototype

```
void setConstraints(Component comp, GridBagConstraints cons);
```

Constraint fields Defined by GridBagConstraints:

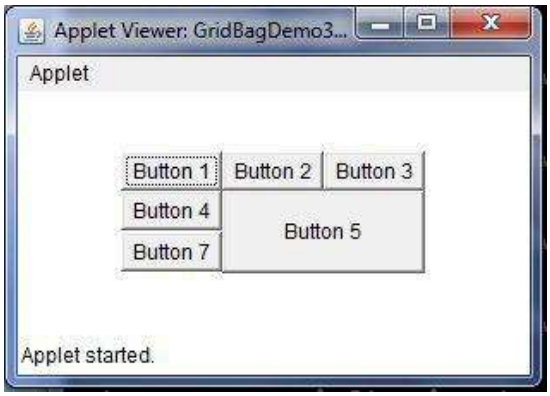
Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER. Others are GridBagConstraints.EAST GridBagConstraints.WEST GridBagConstraints.SOUTH GridBagConstraints.NORTH GridBagConstraints.NORTHEAST GridBagConstraints.NORTHWEST GridBagConstraints.SOUTHEAST GridBagConstraints.SOUTHWEST
int gridx	Specifies the X coordinate of the cell to which the component will be added.
int gridy	Specifies the Y coordinate of the cell to which the component will be added.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
double weightx	Specifies a weight value that determines the horizontal

	spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0. 
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default) GridBagConstraints.HORIZONTAL GridBagConstraints.VERTICAL GridBagConstraints.BOTH.
Insets insets	Small amount of space between the container that holds your components and the window that contains it. Default insets are all zero. Ex. Insets i=new Insets(5,10,20,15); 

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridBagDemo" width=200 height=100>
</applet>
*/
public class GridBagDemo extends Applet
{
        Button b1,b2,b3,b4,b5,b6,b7,b8 ;

        public void init() {
```

```
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);

        // Define check boxes.
        b1=new Button("Button 1");
        b2=new Button("Button 2");
        b3=new Button("Button 3");
        b4=new Button("Button 4");
        b5=new Button("Button 5");
        b6=new Button("Button 6");
        b7=new Button("Button 7");
        b8=new Button("Button 8");

         gbc.gridx=0;
        gbc.gridy=0;
        gbag.setConstraints(b1,gbc);
        gbc.gridx=1;
        gbc.gridy=0;
        gbag.setConstraints(b2,gbc);
        gbc.gridx=2;
        gbc.gridy=0;
        gbag.setConstraints(b3,gbc);
        gbc.gridx=0;
        gbc.gridy=1;
        gbag.setConstraints(b4,gbc);
        gbc.gridx=1;
        gbc.gridy=1;
        gbc.gridwidth=2;
        gbc.gridheight=2;
        gbc.ipady=25;
        gbc.ipadx=20;
        gbc.fill=GridBagConstraints.BOTH;
        gbag.setConstraints(b5,gbc);
        gbc.gridx=0;
        gbc.gridy=2;
        gbc.anchor=GridBagConstraints.WEST;
        gbc.ipady= 0;
        gbc.ipadx= 0;
        gbc.fill=GridBagConstraints.NONE;
        gbag.setConstraints(b7,gbc);
        add(b1);
        add(b2);
        add(b3);
        add(b4);
        add(b5);
        add(b7);

    }
}
```



Swings:

AWT is used for creating GUI in Java. However, the AWT components are internally depends on native methods like C functions and operating system equivalent and hence problems related to portability arise (look and feel. Ex. Windows window and MAC window). And, also AWT components are heavy weight. It means AWT components take more system resources like memory and processor time.

Due to this, Java soft people felt it is better to redevelop AWT package without internally taking the help of native methods. Hence all the classes of AWT are extended to form new classes and a new class library is created. This library is called JFC (Java Foundation Classes).

Java Foundation Classes (JFC):

JFC is an extension of original AWT. It contains classes that are completely portable, since the entire JFC is developed in pure Java. Some of the features of JFC are:

1. JFC components are light-weight: Means they utilize minimum resources.
2. JFC components have same look and feel on all platforms. Once a component is created, it looks same on any OS.
3. JFC offers "pluggable look and feel" feature, which allows the programmer to change look and feel as suited for platform. For, ex if the programmer wants to display window-style button on Windows OS, and Unix style buttons on Unix, it is possible.
4. JFC does not replace AWT, but JFC is an extension to AWT. All the classes of JFC are derived from AWT and hence all the methods in AWT are also applicable in JFC.

So, JFC represents class library developed in pure Java which is an extension to AWT and swing is one package in JFC, which helps to develop GUIs and the name of the package is

```
import javax.swing.*;
```

Here x represents that it is an 'extended package' whose classes are derived from AWT package.

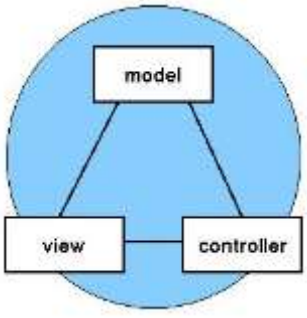
MVC Architecture:

In MVC terminology,

Model corresponds to the state information associated with the component (data).

For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.

The view visual appearance of the component based upon model data.



The controller acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture.

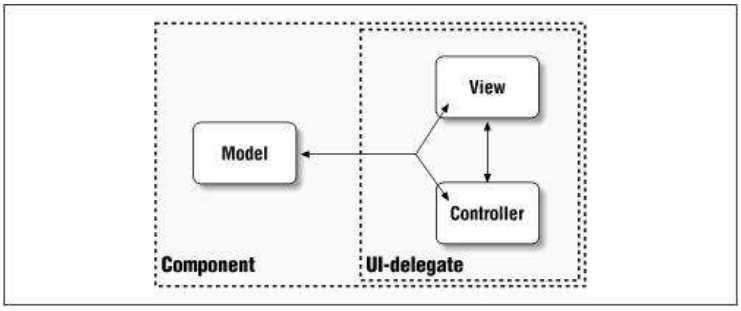


Figure : With Swing, the view and the controller are combined into a UI-delegate object

So let's review: each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate reacts to various events.

Difference between AWT and Swings:

AWT	Swing
Heavy weight	Light weight
Look and feel is OS based	Look and feel is OS independent.
Not pure Java based	Pure Java based
Applet portability: Web-browser is support	Applet portability: A plug-in is required
Do not support features like icon and tool tip.	It supports.
The default layout manager for applet: flow and frame is border layout.	The default layout manger for content pane is border layout.

Components and Containers:

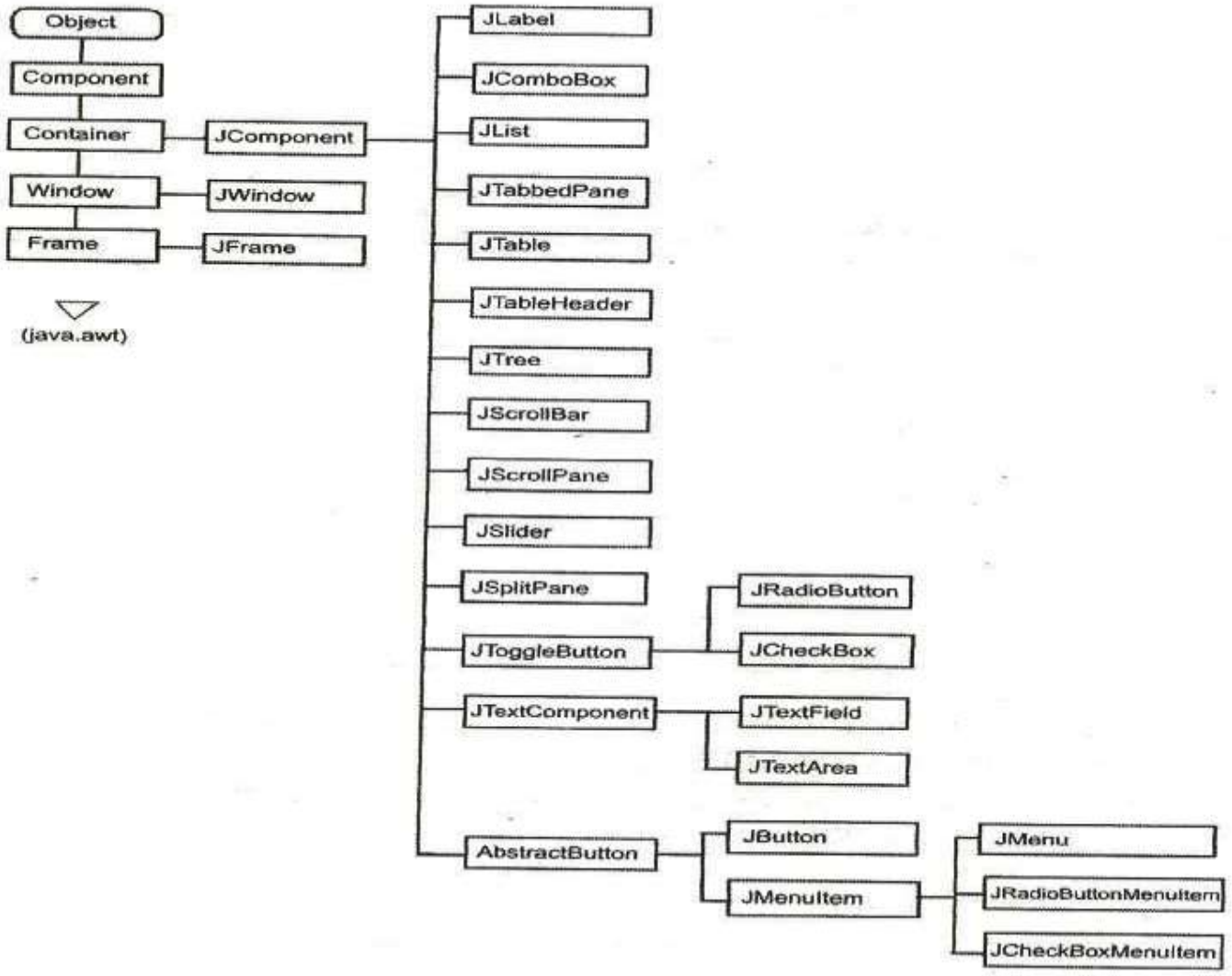
A Swing GUI consists of two key items: ***components*** and ***containers*.**

However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components.

Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, **a container can also hold other containers**. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

Components:

In general, Swing components are derived from the **JComponent** class. **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component.** All of Swing's components are represented by classes defined within the package **javax.swing**. The following figure shows hierarchy of classes of javax.swing.



.

Containers:

Swing defines two types of containers.

1. **Top-level containers/ Root containers:** JFrame, JApplet,JWindow, and JDialog.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications are JFrame and JApplet.

Unlike Swing's other components , the top-level containers are heavyweight. Because they inherit AWT classes Component and Container.

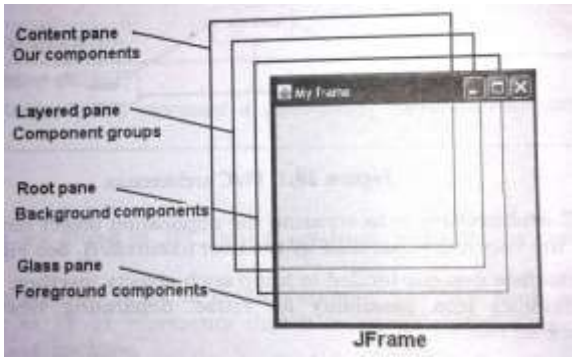
Whenever we create a top level container four sub-level containers are automatically created:

Glass pane (JGlass)

Root pane (JRootPane)

Layered pane(JLayeredPane)

Content pane



Glass pane: This is the first pane and is very close to the monitor's screen. Any components to be displayed in the foreground are attached to this glass pane. To reach this glass pane we use getGlassPane() method of JFrame class, which return Component class object.

Root Pane: This pane is below the glass pane. Any components to be displayed in the background are displayed in this frame. To go to the root pane, we can use getRootPane() method of JFrame class, which returns JRootPane object.

Layered pane: This pane is below the root pane. When we want to take several components as a group, we attach them in the layered pane. We can reach this pane by calling getLayeredPane() method of JFrame class which returns JLayeredPane class object.

Conent pane: This is bottom most of all, Individual components are attached to this pane. To reach this pane, we can call getContentPane() method of JFrame class which returns Container class object.

2. **Lightweight containers** – containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components.

JFrame:

We know frame represents a window with a title bar and borders. Frame becomes the basis for creating the GUIs for an application because all the components go into the frame. To create a fram, we have to create an object to JFrame class in swing as

JFrame jf=new JFrame(); // create a frame without title

JFrame jf=new JFrame("title"); // create a frame with title

To close the frame, use setDefaultCloseOperation() method of JFrame class

setDefaultCloseOperation(constant)

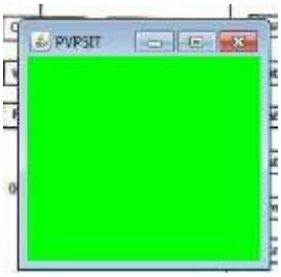
where constant values are

JFrame.EXIT_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DISPOSE_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DO_NOTHING_ON_CLOSE	This will not perform any operation upon clicking close button
JFrame.HIDE_ON_CLOSE	This hides the frame upon clicking close button

Example:


```
import javax.swing.*;
class FrameDemo
{
        public static void main(String arg[])
        {
                JFrame jf=new JFrame("PVPSIT");
                jf.setSize(200,200);
                jf.setVisible(true);
                jf.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE );
        }
}
```

Example: To set the background




```
import javax.swing.*;
 import java.awt.*;
class FrameDemo
{
        public static void main(String arg[])
        {
                JFrame jf=new JFrame("PVPSIT");
                jf.setSize(200,200);
                jf.setVisible(true);
                Container c=jf.getContentPane();
                c.setBackground(Color.green);
        }
}
```

JApplet:

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not found in **Applet**. For example, **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane.

One difference between **Applet** and **JApplet** is, When adding a component to an instance of **JApplet**, do not invoke the **add( )** method of the applet. Instead, call **add( )** for the *content pane* of the **JApplet** object.

The content pane can be obtained via the method shown here:

Container getContentPane( )

The **add( )** method of **Container** can be used to add a component to a content pane. Its form is shown here:

void add(*comp*)

Here, *comp* is the component to be added to the content pane.

JComponent:

The class **JComponent** is the base class for all Swing components except top-level containers. To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level SWING container.

Constructor: JComponent();

The following are the JComponent class's methods to manipulate the appearance of the component.

public int getWidth ()	Returns the current width of this component in pixel.
public int getHeight ()	Returns the current height of this component in pixel.
public int getX()	Returns the current x coordinate of the component's top-left corner.
public int getY ()	Returns the current y coordinate of the component's top-left corner.
public java.awt.Graphics getGraphics()	Returns this component's Graphics object you can draw on. This is useful if you want to change the appearance of a component.
public void setBackground (java.awt.Color bg)	Sets this component's background color.
public void setEnabled (boolean enabled)	Sets whether or not this component is enabled.
public void setFont (java.awt.Font font)	Set the font used to print text on this component.
public void setForeground (java.awt.Color fg)	Set this component's foreground color.
public void setToolTipText(java.lang.String text)	Sets the tool tip text.
public void setVisible (boolean visible)	Sets whether or not this component is visible.

Text Fields

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

JTextField( )
JTextField(int *cols*)
JTextField(String *s*, int *cols*)
JTextField(String *s*)

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field.

The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a **JTextField** object is created and is added to the content pane.

Example:

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
        JTextField jtf;
        public void init()
        {
                // Get content pane
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());
                // Add text field to content
                pane jtf = new JTextField(10);
                contentPane.add(jtf);
        }
}
```

The JButton Class

The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

JButton(Icon *i*)
JButton(String *s*)
JButton(String *s*, Icon *i*)
Here, *s* and *i* are the string and icon used for the button.

Example:

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

```
/*
<applet code="JButtonDemo2" width=250 height=300>
</applet>
*/
public class JButtonDemo2 extends JApplet implements ActionListener
{
        JTextField jtf;
        public void init()
        {
                // Get content pane
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());

                JButton jb1 = new JButton("BEC");
                jb1.addActionListener(this);
                contentPane.add(jb1);

                // Add buttons to content pane
                ImageIcon pvp = new ImageIcon("pvp.jpg");
                JButton jb2 = new JButton("PVPSIT",pvp);
                jb2.setActionCommand("PVPSIT");
                jb2.addActionListener(this);
                contentPane.add(jb2);

                jtf = new JTextField(10);
                contentPane.add(jtf);
        }
        public void actionPerformed(ActionEvent ae)
        {
                jtf.setText(ae.getActionCommand());
        }
}
```

Check Boxes:

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**. Its immediate super class is **JToggleButton**, which provides support for two-state buttons (true or false). Some of its constructors are shown here:

JCheckBox(Icon *i*)
JCheckBox(Icon *i*, boolean *state*)
JCheckBox(String *s*)
JCheckBox(String *s*, boolean *state*)
JCheckBox(String *s*, Icon *i*)
JCheckBox(String *s*, Icon *i*, boolean *state*)

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not.

The state of the check box can be changed via the following method:

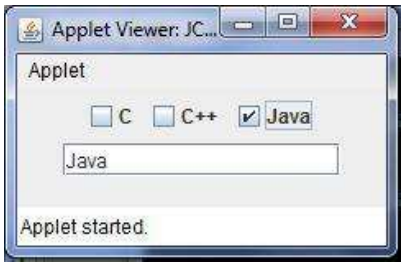
void setSelected(boolean *state*)

Here, *state* is **true** if the check box should be checked.

When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged( )**. Inside **itemStateChanged( )**, the **getItem( )** method gets the **JCheckBox** object that generated the event. The **getText( )** method gets the text for that check box and uses it to set the text inside the text field.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo2" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo2 extends JApplet implements ItemListener
{
    JTextField jtf;
    public void init() {
        // Get content pane
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JCheckBox cb = new JCheckBox("C", true);
        cb.addItemListener(this);
        contentPane.add(cb);

        cb = new JCheckBox("C++");
        cb.addItemListener(this);
        contentPane.add(cb);

        cb = new JCheckBox("Java");
        cb.addItemListener(this);
        contentPane.add(cb);

        // Add text field to the content
        pane jtf = new JTextField(15);
        contentPane.add(jtf);
    }
    public void itemStateChanged(ItemEvent ie) {
        JCheckBox cb = (JCheckBox)ie.getItem();
        jtf.setText(cb.getText());
    }
}
```



Radio Buttons:

Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are shown here:

JRadioButton(Icon *i*)
JRadioButton(Icon *i*, boolean *state*)
JRadioButton(String *s*)
JRadioButton(String *s*, boolean *state*)
JRadioButton(String *s*, Icon *i*)
JRadioButton(String *s*, Icon *i*, boolean *state*)

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button is initially selected. Otherwise, it is not.

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton *ab*)

Here, *ab* is a reference to the button to be added to the group.

Radio button presses generate action events that are handled by **actionPerformed( )**. The **getActionCommand( )** method gets the text that is associated with a radio button and uses it to set the text field.

Example:


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet implements ActionListener
{
    JTextField tf;
    public void init()
    {
        // Get content pane
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        // Add radio buttons to content pane
        JRadioButton c = new JRadioButton("C");
        c.addActionListener(this);
        contentPane.add(c);
```

```
        JRadioButton cpp = new JRadioButton("C++");
        cpp.addActionListener(this);
        contentPane.add(cpp);

        JRadioButton java = new JRadioButton("JAVA");
        java.addActionListener(this);
        contentPane.add(java);

        // Define a button group
        ButtonGroup bg = new ButtonGroup();
        bg.add(c);
        bg.add(cpp);
        bg.add(java);

        // Create a text field and add it to the content pane
        tf = new JTextField(5); contentPane.add(tf);

    }
    public void actionPerformed(ActionEvent ae)
        { tf.setText(ae.getActionCommand());
    }
}
```

Combo boxes:

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.

A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.

Two of **JComboBox**'s constructors are shown here:

JComboBox( )
JComboBox(Vector *v*)

Here, *v* is a vector that initializes the combo box. Items are added to the list of choices via the **addItem( )** method, whose signature is shown here:

void addItem(Object *obj*)

Here, *obj* is the object to be added to the combo box.

By default, a JComboBox component is created in read-only mode, which means the user can only pick one item from the fixed options in the drop-down list. If we want to allow the user to provide his own option, we can simply use the setEditable() method to make the combo box editable.

The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for "PVPSIT", "BEC", and "VRSEC". When a college is selected, the label is updated to display the flag for that country

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/* <applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet implements ItemListener
{
        Container contentPane;

        public void init()
        {
                // Get content pane
                contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());

                // Create a combo box and add it to the
                panel JComboBox jc = new JComboBox();
                jc.addItem("pvp");
                jc.addItem("bec");
                jc.addItem("vrsec");

                jc.addItemListener(this);
                contentPane.add(jc);

         }
        public void itemStateChanged(ItemEvent ie)
        {
                String s = (String)ie.getItem();
                JOptionPane.showMessageDialog(null,"You selected:"+s);
        }
}
```

Tabbed Panes:

A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:

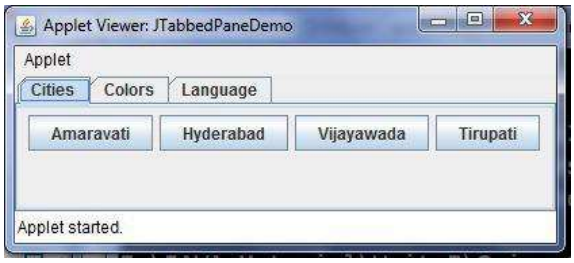
void addTab(String *str*, Component *comp*)

Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.

The general procedure to use a tabbed pane in an applet is outlined here:

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

The following example illustrates how to create a tabbed pane. The first tab is titled"Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Language" and contains radio buttons.



Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet
{
	public void init()
	{
		Container contentPane = getContentPane();
		JTabbedPane jtp = new JTabbedPane();
		jtp.addTab("Cities", new CitiesPanel());
		jtp.addTab("Colors", new ColorsPanel());
		jtp.addTab("Language", new LanguagesPanel());
		contentPane.add(jtp);
	}
}
class CitiesPanel extends JPanel
{
	public CitiesPanel()
	{
		JButton b1 = new JButton("Amaravati");
		add(b1);
		JButton b2 = new JButton("Hyderabad");
		add(b2);
		JButton b3 = new JButton("Vijayawada");
		add(b3);
		JButton b4 = new JButton("Tirupati");
```

```
                add(b4);
        }
}
class ColorsPanel extends JPanel
{
        public ColorsPanel()
        {
                JCheckBox cb1 = new JCheckBox("Red");
                add(cb1);
                JCheckBox cb2 = new JCheckBox("Green");
                add(cb2);
                JCheckBox cb3 = new JCheckBox("Blue");
                add(cb3);
        }
}
class LanguagesPanel extends JPanel
{
        public LanguagesPanel()
        {
                JRadioButton rb1 = new JRadioButton("Telugu");
                add(rb1);
                JRadioButton rb2 = new JRadioButton("Hindi");
                add(rb2);
                JRadioButton rb3 = new JRadioButton("English");
                add(rb3);
        }
}
```

JLabel :

JLabel is a class of java Swing . JLabel is used to display a short string or an image icon. JLabel can display text, image or both . JLabel is only a display of text or image and it cannot get focus . JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centered but the user can change the alignment of label.

Constructor of the class are :

1. **JLabel() :** creates a blank label with no text or image in it.
2. **JLabel(String s) :** creates a new label with the string specified.
3. **JLabel(Icon i) :** creates a new label with a image on it.
4. **JLabel(String s, Icon i, int align) :** creates a new label with a string, an image and a specified horizontal alignment

Commonly used methods of the class are :

1. **getIcon() :** returns the image that the label displays
2. **setIcon(Icon i) :** sets the icon that the label will display to image i
3. **getText() :** returns the text that the label will display
4. **setText(String s) :** sets the text that the label will display to string s

JList :

JList is part of Java Swing package . JList is a component that displays a set of Objects and allows the user to select one or more items . JList inherits JComponent class. JList is a easy way to display an array of Vectors .

Constructor for JList are :

1. **JList()**: creates an empty blank list
2. **JList(E [ ] l)** : creates an new list with the elements of the array.
3. **JList(ListModel d)**: creates a new list with the specified List Model
4. **JList(Vector l)** : creates a new list with the elements of the vector

Commonly used methods are :

method	explanation
getSelectedIndex()	returns the index of selected item of the list
getSelectedValue()	returns the selected value of the element of the list
setSelectedIndex(int i)	sets the selected index of the list to i
getSelectedValuesList()	returns a list of all the selected items.
getSelectedIndices()	returns an array of all of the selected indices, in increasing order

```
// java Program to create a simple JList

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


class solve extends JFrame
{
```

```
    //frame
    static JFrame f;

    //lists
    static JList b;


    //main class
    public static void main(String[] args)
    {
        //create a new frame
        f = new JFrame("frame");

        //create a object
        solve s=new solve();

        //create a panel
        JPanel p =new JPanel();

        //create a new label
        JLabel l= new JLabel("select the day of the week");

        //String array to store weekdays
        String week[]= { "Monday","Tuesday","Wednesday",
                        "Thursday","Friday","Saturday","Sunday"};

        //create list
        b= new JList(week);

        //set a selected index
        b.setSelectedIndex(2);

        //add list to panel
        p.add(b);

        f.add(p);

        //set the size of frame
        f.setSize(400,400);

        f.show();
    }


}
```

APPLETS

A Java applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run. An applet is typically embedded inside a web page and runs in the context of a browser. An applet must be a subclass of the **java.applet.Applet** class. The Applet class provides the standard interface between the applet and the browser environment.

The **Applet** class is contained in the **java.applet** package.**Applet** contains several methods that give you detailed control over the execution of your applet.

In addition,**java.applet** package also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Applet Basics:

All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**. Applets must also import **java.awt**. **AWT** stands for the Abstract Window Toolkit. Since all applets run in a window, it is necessary to include support for that window by importing java.awt package.

Applets are not executed by the console-based Java run-time interpreter. Rather, they are executed by either a Web browser or an applet viewer.

Execution of an applet does not begin at **main( ).** Output to your applet's window is not performed by **System.out.println( )**. Rather, it is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location. Input is also handled differently than in an application.

Once an applet has been compiled, it is included in an HTML file using theAPPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file.
To view and test an applet more conveniently, simply include a comment at the head of your
Java source code file that contains the APPLET tag.

Here is an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this tutorial will contain the appropriate APPLET tag embedded in a comment.

The Applet Class:

Applet extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities

Applet Architecture:

An applet is a window-based program. As such, its architecture is different from the so-called normal, console-based programs .

First, applets are event driven. it is important to understand in a general way how the event- driven architecture impacts the design of an applet.

Here is how the process works. An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the AWT.

Applet Initialization and Termination(Applet Life Cycle methods):

It is important to understand the order in which the various methods shown in theskeleton are called. When an applet begins, the AWT calls the following methods, in this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

init( ):init( ) method is called once—the first time an applet is loaded. The **init( )** method is the first method to be called. This is where you should initialize variables.

start(): The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped(i.e start() method is called every time, the applet resumes execution).

Paint(): The **paint( )** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The
paint( ) method has one parameter of type **Graphics**.

stop( ): The stop() method is called when the applet is stopped(i.e for example ,when the applet is minimized the stop method is called).

destroy( ): The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory(i.e destroy() method is called when the applet is about to terminate).The **stop( )** method is always called before **destroy( )**.

An Applet Skeleton Program:

Four methods—**init( )**, **start( )**, **stop( )**, and **destroy( )**—are defined by **Applet**. Another, **paint( )**, is defined by the AWT **Component** class. All applets must import **java.applet**. Applets must also import **java.awt.**

These five methods can be assembled into the skeleton shown here:

```
// An Applet
skeleton. import
java.awt.*; import
java.applet.*;
/
*
<applet code="AppletSkel" width=300
height=100>
</app
let>
*
/
public class AppletSkel extends
Applet
{
  // Called
  first. public
  void init()
 {
    // initialization
 }
/* Called second, after init(). Also called
    whenever the applet is restarted. */
public void start()
{
   // start or resume execution
}
// Called when the applet is
stopped. public void stop()
{
   // suspends execution}
  /* Called when applet is terminated. This is the
  last method executed. */
  public void destroy()
 {
   // perform shutdown activities
 }
  // Called when an applet's window must be
  restored. public void paint(Graphics g)
 {
   // redisplay contents of window
```

```
  }
}
```

Simple Applet programe:

SimpleApplet.java

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=300 height=100>
</applet>
*/

public class SimpleApplet extends Applet
{
   String msg="";
   // Called first.
  public void init()
  {
     msg="Hello";
  }
   /* Called second, after init().
   Also called      whenever  the applet is restarted. */
  public void start()
  {
     msg=msg+",Welcome to Applet";
  }

  // whenever the applet must redraw its output, paint( ) is
  called. public void paint(Graphics g)
  {

     g.drawString(msg,20,20);
  }
}
```

Output:

How To Run an Applet Programe:

There are two ways in which you can run an applet:

■ Executing the applet within a Java-compatible Web browser.

■ Using an applet viewer, such as the standard SDK tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastestand easiest way to test your applet.

Using an applet viewer to run applet(demonstrates you to run SimpleApplet.java):

Place the applet tag in comments in java source code.

Note:Code attribute value must be equal to name of class which extends Applet class.

Compiling: javac SimpleApplet.java

Run: AppletViewer SimpleApplet.java

Executing the applet within a Java-compatible Web browser(demonstrates you to run SimpleApplet.java):

Compiling: javac SimpleApplet.java

Create an Html file and embeded Applet tag in html file.

Attributes in applet tag:

Code(attribute):specify name of applet class to load into browser.

Width(attribute):width of an applet.

Height(attribute):height of an applet.

SimpleApplet.html

```
<html>
  <body>
    <applet code="SimpleApplet" width=300 height=100></applet>
  </body>
</html>
```

When you open SimpleApplet.html , SimpleApplet.class applet is loaded into browser.

Note: The Browser must be java enabled to load applet programe.

Simple Applet Display Methods:

As we've mentioned, applets are displayed in a window and they use the AWT to perform input and output.To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class.Graphics class is defined in **java.awt** package.

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output and x and y are x-coordinate ,y-coordinate respectively. In a Java window, the upper-left corner is location 0,0.

To set the background color of an applet's window, use **setBackground( )**. To set the foreground color (the color in which text is shown, for example), use **setForeground( )**. These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors:

Color.black Color.magenta
Color.blue Color.orange
Color.cyan Color.pink
Color.darkGray Color.red
Color.gray Color.white
Color.green Color.yellow
Color.lightGray

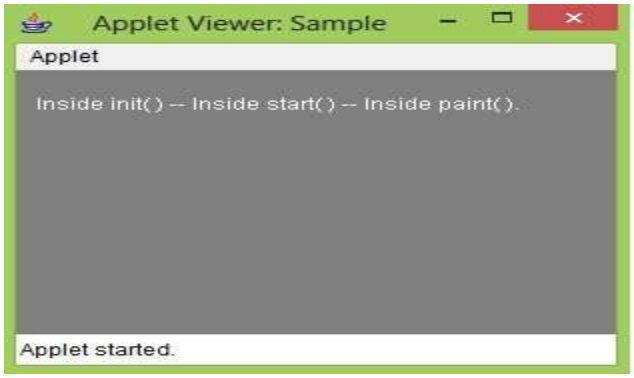
For example, this sets the background color to green and the text color to red:

setBackground(Color.green);
setForeground(Color.red);

Sample.java

```
/* A simple applet that sets the foreground and background colors and outputs a string.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=200>
</applet>
*/
 public class Sample extends Applet
 {
    String msg;
      public void init()
    {
      setBackground(Color.gray);
      setForeground(Color.white);
      msg = "Inside init( ) --";
    }
    // Initialize the string to be
    displayed. public void start()
    {
      msg += " Inside start( ) --";
    }
    // Display msg in applet window.
   public void paint(Graphics g)
   {   msg += " Inside paint( ).";
       g.drawString(msg, 10, 30);
   }
}
```

Outp
ut:



RequestingRepainting:

Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**.

The simplest version of **repaint( )** is shown here:

void repaint( )

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

void repaint(int *left*, int *top*, int *width*, int *height*)

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels. You save time by specifying a region to repaint.

The other two versions of repaint():

void repaint(long *maxDelay*)

void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called.

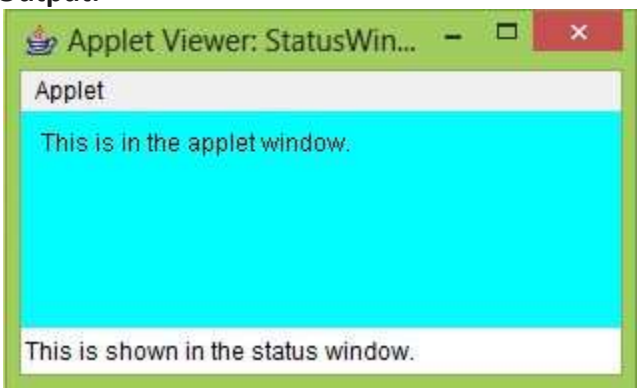
Using the Status window:

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )** with the string that you want displayed.

```
// Using the Status
Window. import
java.awt.*;
import
java.applet.*;
/
*
```

```
<applet code="StatusWindow" width=300
height=300>
</appl
et>
*
/
public class StatusWindow extends
Applet
{
  public void init()
  {setBackground(Color.cyan);
  }
  // Display msg in applet window.
  public void paint(Graphics g)
  {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus("This is shown in the status window.");
  }
}
```

Output:



Types of applets : (Based on look and Feel)

There are two varieties of applets. The first are those based directly on the **Applet** class. These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. **JApplet** inherits **Applet**, all the features of **Applet** are also available in **Japplet**.

Types of Applets (In General) :

Web pages can contain two types of applets which are named after the location at which they are stored.

1. Local Applet
2. Remote Applet

Local Applets: A local applet is the one that is stored on our own computer system. When the Web-page has to find a local applet, it doesn't need to retrieve information from the Internet. A local applet is specified by a path name and a file name as shown below in which the codebase attribute specifies a path name, whereas the code attribute specifies the name of the byte-code file that contains the applet's code.

<applet codebase="MyAppPath" code="MyApp.class" width=200 height=200> </applet>

Remote Applets: A remote applet is the one that is located on a remote computer system . This computer system may be located in the building next door or it may be on the other side of the world. No matter where the remote applet is located, it's downloaded onto our computer via the Internet. The browser must be connected to the Internet at the time it needs to display the remote applet. To reference a remote applet in Web page, we must know the applet's URL (where it's located on the Web) and any attributes and parameters that we need to supply. A local applet is specified by a url and a file name as shown below.

<applet codebase="http://www.apoorvacollege.com" code="MyApp.class" width=200 height=200> </applet>

JAVA LAB MANUAL

EXERCISE-1

a) Write a Java program to print largest of three numbers.

```
class Largest
{
 public static void main(String args[])
  {
    int a,b,c;
    a=Integer.parseInt(args[0]);
    b=Integer.parseInt(args[1]);
    c=Integer.parseInt(args[2]);
     if(a>b&&a>c)
        System.out.println(a +" is largest");
     else if(b>a&&b>c)
        System.out.println(b +" is largest");
      else
       System.out.println(c +" is largest");
  }
}
```

OUTPUT:

```
C:\java>javac Largest.java

C:\java>java Largest 12 34 15
34 is largest
```

b) Write a java program to calculate the sum of all the numbers divisible by 3 from 1 to n and print the sum.

```
class Sum
{
  public  static void main(String args[])
   {
      int i,n,sum=0;
      n=Integer.parseInt(args[0]);
       for(i=1;i<n;i++)
        {
          if(i%3==0)
           {
sum=sum+i;
           }
        }
      System.out.println("Sum of numbers divisible by 3 between 1 and "+n+" is :");
System.out.println(sum);
   }
}
```

OUTPUT:

```
C:\java>javac Sum.java

C:\java>java Sum 15
Sum of numbers divisible by 3 between 1 and 15 is :
30
```

c) Write a java program to calculate the sum of first "n" even integer numbers and "n" odd integer numbers excluding 0.

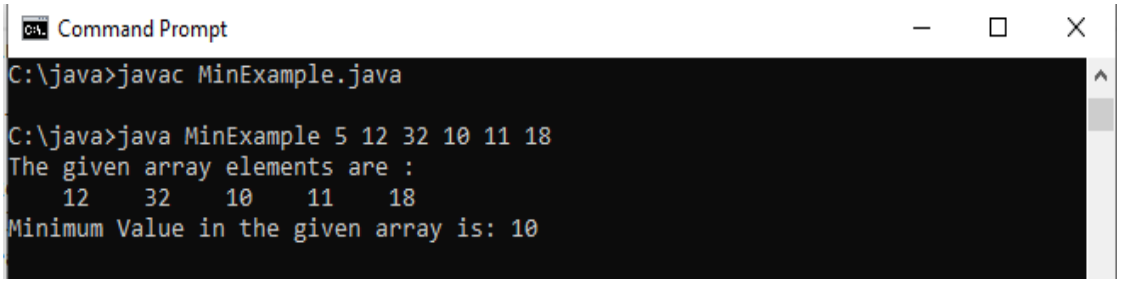
```
class SumofInts
{
 public static void main(String args[])
 {
  int i,n,esum=0,osum=0;
  n=Integer.parseInt(args[0]);
   for(i=1;i<n;i++)
    {
     if(i%2==0)
      {
       esum=esum+i;
      }
     else
     {
      osum=osum+i;
     }
    }
     System.out.println("sum of even numbers between 1 and"+n+ " :"+ esum);
    System.out.println("sum of odd numbers between 1 and"+n+ " :"+ osum);
  }
}
```

OUTPUT:

d) Write a java program to read the size of an array from keyboard. You have to initialize the array and insert the elements into it. You have to find the minimum number in that array and print the same.

```
class MinExample
  {
   public static void main(String args[])
  {
    int size;
    size = Integer.parseInt(args[0]);
    int array[] = new int[size];
    int min = 0 ;
    System.out.println("The given array elements are : ");
    for(int i=0;i<size;i++)
    {
      array[i] = Integer.parseInt(args[i+1]);
      System.out.print( "    " + array[i]);
     }
    min = array[0];
    for(int i=0;i<size;i++)
    {
         if(array[i] < min)
        {
         min = array[i];
        }
    }
    System.out.println("Minimum Value in the given array is: "+ min);
  }
   }
```

OUTPUT:

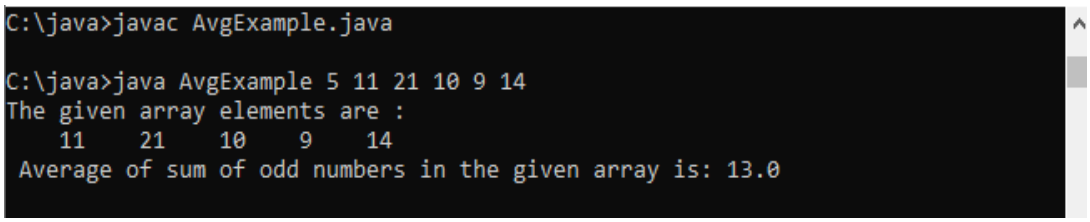


e) Write a java program to find the average of all odd numbers present in the array and print the same.

```
class AvgExample
 {
    public static void main(String args[])
   {
     int size;
     size = Integer.parseInt(args[0]);
     int array[] = new int[size];
     int osum =0;
     int count =0;
     float avg = 0 ;
     System.out.println("The given array elements are : ");
     for(int i=0;i<size;i++)
     {
      array[i] = Integer.parseInt(args[i+1]);
      System.out.print( "    " + array[i]);
     }

     for(int i=0;i<size;i++)
     {
          if((array[i] % 2)!= 0)
         {
          count = count + 1;
          osum = osum + array[i];
```

```
        }
      }
       avg = osum / count ;
      System.out.println("\n Average of sum of odd numbers in the given array is: " +  avg);
   }
}
```

OUTPUT:



EXERCISE-2

a) Implement a program by using the concept of returning value from a method.

```
classReturn
{
publicstaticvoidmain(Stringargs[])
{
booleant=true; System.out.println("Beforethereturn.");
if(t)
return;//returnto caller
System.out.println("Thiswon'texecute.");
}
}
```

OUTPUT:

b) Implement a program to demonstrate constructors.

```
import java.util.Scanner;
class Box
{
        double length;
    double breadth;
    double height;

        Box()
        {
                Scanner s=new Scanner(System.in);
                System.out.println("Enter length,breadth and height values");
                length=s.nextDouble();
```

```
                breadth=s.nextDouble();
                height=s.nextDouble();
        }
    double calvolume ()
    {
     return length * breadth * height;
        }
}
class BoxDemo
{
        public static void main(String arj[])
        {
                double vol;
          Box b=new Box();
          vol=b.calvolume();
                System.out.println("Volume of the given box is:" + vol);

        }
    }
```

OUTPUT:

```
C:\java>javac BoxDemo.java

C:\java>java BoxDemo
Enter length,breadth and height values
5
4
8
Volume of the given box is:160.0
```

c) Implement a program to demonstrate the concept of method overloading.

```
class Sum
 {
   // Overloaded sum() , This sum takes two int parameters
   public int sum(int x, int y)
   {
      return (x + y);
```

```
    }
    // Overloaded sum(), This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }
    // Overloaded sum(), This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }
  }
    class SumTest
  {
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println("Sum of Two Intergers:" +  s.sum(10, 20));
        System.out.println("Sum of Three Intergers:" +  s.sum(10, 20, 30));
        System.out.println("Sum of  Doubles :" +  s.sum(10.5, 20.5));
    }
}
```

```
C:\java>javac SumTest.java

C:\java>java SumTest
Sum of Two Intergers:30
Sum of Three Intergers:60
Sum of  Doubles :31.0
```

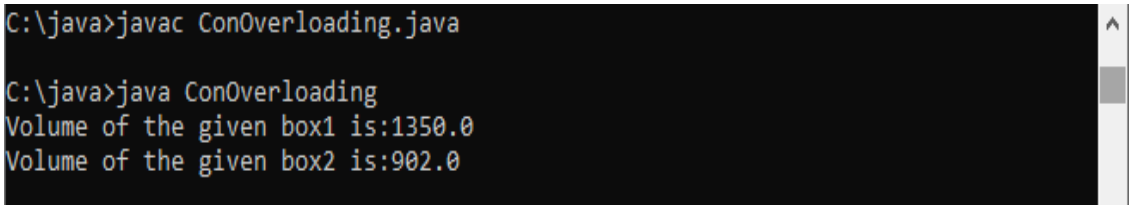
OUTPUT:

d) Implement a program to demonstrate Constructor Overloading.

```
import java.util.Scanner;
class SampleBox
{
        double length;
        double breadth;
         double height;

        SampleBox()
        {
                length=15;
          breadth=12;
          height=7.5;
        }
     SampleBox(double l,double b,double h)
        {
          length= l;
           breadth= b;
           height = h;
        }
     double calvolume ()
     {
      return length * breadth * height;
        }
}
class ConOverloading
{
        public static void main(String arg[])
        {
                double vol1, vol2;
          SampleBox b=new SampleBox();
          SampleBox b1=new SampleBox(10.0,16.4,5.5);
```

```
        vol1=b.calvolume();
                System.out.println("Volume of the given box1 is:" + vol1);
        vol2=b1.calvolume();
                System.out.println("Volume of the given box2 is:" + vol2);


    }
}
```

OUTPUT:



e) Implement a program to pass object as parameter.

```
class Test
 {
 int a,b;
Test(int i,int j)
{
 a=i;
 b=j;
  }
     boolean equals(Test 0)
      {
           if(o.a==a&&a.b==b )
            return true;
             else
              return false;
      }
    }

 class PassObj
```

```
  {
public static void main(String args[])
        {
            Test ob1=new Test(100,22);
             Test ob2=new Test(100,22);
             Test ob3=new Test(1,1);
              System.out.println("ob1==ob2: "+ob1.equals(ob2));
              System.out.println("ob1==ob3: "+ob1.equals(ob3));
         }
    }
```

OUTPUT:

```
C:\Users\HP\Desktop\Java>javac PassObj.java

C:\Users\HP\Desktop\Java>java PassObj
ob1==ob2: true
ob1==ob3: false
```

EXERCISE-3

a) Write a program using the concept of String class and its methods.

```
class String_Demo
{
public static void main(String arg[])
{
String s1="PVP Siddhartha";
String s2=new String("Engineering");
String s3=new String("College");
String s4 = "    Hello    ";
System.out.println("First :"+s1);
System.out.println("Second :"+s2);
System.out.println("Third:"+s3);
System.out.println("Length of first String :"+s1.length());
System.out.println("Concatenation of first and second :"+s1.concat(s2));
System.out.println("Concatenation of strings with + :"+s1+" "+s3);
System.out.println("If string s1 starts with P or not :"+s1.startsWith("P"));
System.out.println("Extraction of substrings :"+s1.substring(2));
System.out.println("Extraction of substrings:"+s1.substring(2,5));
System.out.println("String into uppercase:"+s1.toUpperCase());
System.out.println("String into lowercase:"+s1.toLowerCase());
System.out.println(" Use of Trim :" + s4.trim());
}
}
```

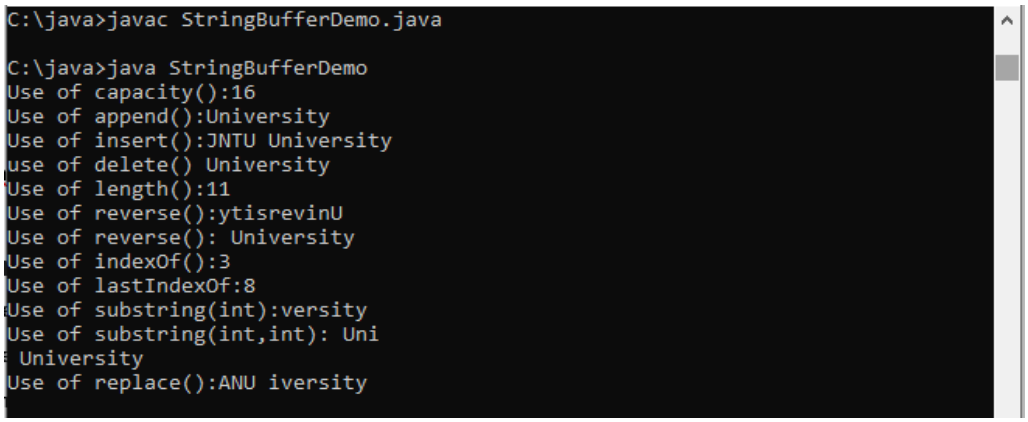
OUTPUT:

```
C:\java>javac String_Demo.java

C:\java>java String_Demo
First :PVP Siddhartha
Second :Engineering
Third:College
Length of first String :14
Concatenation of first and second :PVP SiddharthaEngineering
Concatenation of strings with + :PVP Siddhartha College
If string s1 starts with P or not :true
Extraction of substrings :P Siddhartha
Extraction of substrings:P S
String into uppercase:PVP SIDDHARTHA
String into uppercase:pvp siddhartha
 Use of Trim :Hello
```

b) Implement a program using the concept of String Buffer and its methods.

```
class StringBufferDemo
{
    public static void main(String arg[])
    {
     StringBuffer sb=new StringBuffer();
    System.out.println("Use of capacity():"+sb.capacity());
     StringBuffer sb1=new StringBuffer("Uni");
    System.out.println("Use of append():"+sb1.append("versity"));
    System.out.println("Use of insert():"+sb1.insert(0, "JNTU "));
    System.out.println("use of delete()"+sb1.delete(0,4));
    System.out.println("Use of length():"+sb1.length());
    System.out.println("Use of reverse():"+sb1.reverse());
    System.out.println("Use of reverse():"+sb1.reverse());
    System.out.println("Use of indexOf():"+sb1.indexOf("i"));
    System.out.println("Use of lastIndexOf:"+sb1.lastIndexOf("i"));
    System.out.println("Use of substring(int):"+sb1.substring(4));
    System.out.println("Use of substring(int,int):"+sb1.substring(0,4));
    System.out.println(sb1);
    System.out.println("Use of replace():"+sb1.replace(0,3,"ANU "));
    }
}
```

OUTPUT:



c) Write a Java Program that reads a line of integers, and then display each integer, and the sum of all the integers using StringTokenizer class and it's methods.

```
import java.util.*;
 class StringTokenizerEx
{
public static void main(String[] args)
{
 Scanner s = new Scanner(System.in);
System.out.print("\nEnter A Line Of Integers:");
 String line = s.nextLine();
 StringTokenizer st = new StringTokenizer(line);
System.out.println("Number of tokens : "+ st.countTokens());
int sum = 0;
System.out.print("Tokens are : " );
while (st.hasMoreTokens())
{
int i = Integer.parseInt(st.nextToken());
System.out.println(i);
sum = sum + i;
}
System.out.println("The Sum Is :" +sum);
}
}
```

OUTPUT:

```
C:\java>javac StringTokenizerEx.java

C:\java>java StringTokenizerEx

Enter A Line Of Integers:32 3 12 14 16 23
Number of tokens : 6
Tokens are : 32
3
12
14
16
23
The Sum Is :100
```

EXERCISE-4

a) Implement a program using the concept of method overloading.

```
class Human
{
public void eat()
{
   System.out.println("Human is eating");
}
}
class Boy extends Human
{
//Overriding method
public void eat()
{    System.out.println("Boy is eating");
}
public static void main( String args[]) {
   Boy obj = new Boy();
   //This will call the child class version of eat()
   obj.eat();
 }
}
```

OUTPUT:

b) Implement a program using the concept of dynamic method dispatch.

```
class Figure
{
double dim1;
double dim2;
Figure(double a, double b)
 {
   dim1 = a;
   dim2 = b;
  }
double area()
{
System.out.println("Area for Figure is undefined.");
return 0;
}
}
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 FindAreas
{
public static void main(String args[])
 {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
 figref = t;
```

```
System.out.println("Area is " + figref.area());

figref = f;

System.out.println("Area is " + figref.area());

}

}
```

OUTPUT:

c) Implement a program using the concept of Abstract class.

```
abstract class A
{
abstract void callme();
    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();

    }

}
```

OUTPUT:

d) Implement a program using the concept of Final in inheritance.

```
final class Vehicle
{
void run()
{
System.out.println("running");
}
}
class Bike extends Vehicle
{
void run()
{
System.out.println("running safely with 100kmph");
}
}
class FinalClass
{
```

```
public static void main(String args[])
{
Bike b= new Bike();
b.run();
}
}
```

OUTPUT:if final keyword is removed.

EXERCISE-5

a) Write a program using the concept of implementing interfaces.

```
 interface Iface
{
  void meth();
}
 class Simple implements Iface
{
  public void meth()
  {
   System.out.println(" This is an interface Method");
   }
}
class IfaceTest
{
  public static void main(String args[])
  {
   Simple ob =new Simple();
   ob.meth();
   }
}
```

OUTPUT:

```
C:\java>javac IfaceTest.java

C:\java>java IFaceTest
 This is an interface Method
```

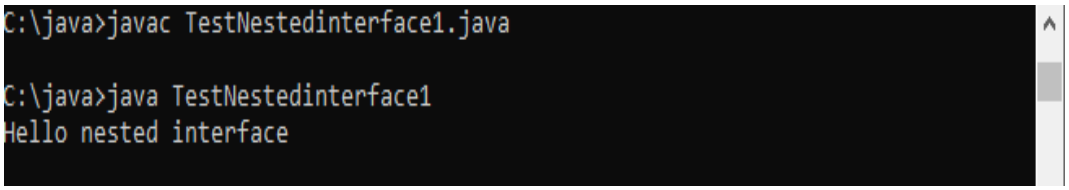
b) Implement a program using the concept of Nested Interfaces.

(i)Using interfaces

```
interface  Showable
{
  void show();
            interface Message
{
   void msg();
```

```
 }
}
          class TestNestedinterface1  implements  Showable.Message
{
 public void msg()
{
 System.out.println("Hello nested interface");
}
   public static void main(String args[])
{
  Showable.Message m;
TestNestedInterface1 ob=new TestNestedInterface1();//upcasting here
 m=ob;      m.msg();
 }
     }
```

OUTPUT:



(ii)Using Class

```
class A
 {
  interface Message
  {
   void msg();
  }
}
```

```
Class NestedIntTest2 implements A.Message
{
 public void msg()
{
System.out.println("Hello nested interface");
}
   public static void main(String args[])
 {
  A.Message message=new NestedIntTest2();//upcasting here
  message.msg();
 }
}
```

OUTPUT:

```
C:\java>javac NestedIntTest2.java

C:\java>java NestedIntTest2
Hello nested interface
```

c)Implement a program using the concept of Interface reference.

```
interface Test
{
void call();
}
class InterfaceTest implements Test
 {
public void call()
{
System.out.println("call method called");
}
}
 class IfaceRef
{
```

```
public static void main(String[] args)
{
Test f ;
InterfaceTest it= new InterfaceTest();
f=it;
f.call();
}
}
```

OUTPUT:

---

d)Implement a program using the concept of extending interfaces.

```
interfaceTeacher
  {
   voiddisplay1();
   }
 interfaceStudent
   {
    voiddisplay2();
   }
  interfaceT_SextendsTeacher,Student
   {
    voiddisplay3();
   }
   classCollegeimplementsT_S
    {
      publicvoiddisplay1()
    {
       System.out.println("HiIamTeacher");
    }
```

```
        publicvoiddisplay2()
        {
        System.out.println("HiIamStudent");
         }
        publicvoiddisplay3()
        {
         System.out.println("HiIamTeacher_Student");
         }
        }
       classClassInterface
         {

publicstaticvoidmain(Stringarh[])
  {
Collegec=newCollege();
c.display1();
c.display2();
 c.display3();
      }
   }
```

OUTPUT:

EXERCISE-6

a) Create a user defined package and demonstrate different ways of importing packages.

(a)Creating Addition1 class under package ‘pack’

```
package pack;
public class Addition1
{
int x,y;
public Addition1(int a, int b)
{
x=a;
y=b;
}
public void sum()
{
System.out.println("Sum :"+(x+y));
}
}
```

(b) Creating Subtraction1 class under package ‘pack’

```
package pack;
public class Subtraction1
{
int x,y;
public Subtraction1(int a, int b)
{
x=a;
y=b;
}
```

```
public void difference()
{
System.out.println("Difference :"+(x-y));
}
}
```

Acessing a User defined package

(I)Using import(package name and class name)

```
import pack.Addition1;
import pack.Subtraction1;
class PackTest
{
public static void main(String arg[])
{
Addition1 a=new  Addition1(10,15);
a.sum();
Subtraction1 s=new Subtraction1(20,15);
s.difference();
}
}
```

OUTPUT:

(II)using import (Package name and wildcard '*')

```
import pack.*;
        class PackTesting
       {
        public static void main(String arg[])
       {
         Add a=new  Add(10,15);
         a.sum();
         Sub s=new Sub(20,15);
         s.diff();
       }
        }
```

OUTPUT:

(III) Using Fully qualified name

```
Class PackTesting2
{
public static void main(String args[])
{
pack.Addition1a = new Addition1(10,20);
a.sum();
pack.Subtraction1 s = new Subtraction1(20,5);
s.difference();
}
```

}

OUTPUT:

(b) Implement a program using the concept of

(I)Multiple catch clauses

```
class MultipleCatches
 {
 public static void main(String args[])    {
  try
    {
 int a = args.length;
   System.out.println("a = " + a);
   int b = 42 / a;
 int c[] = { 1 };
 c[42] = 99;
     }
  catch(ArithmeticException e)
   {
 System.out.println("Divide by 0: " + e);
   }
 catch(Exception e)
  {
 System.out.println("Array index out of bounds: " + e);
   }
 System.out.println("After try/catch blocks.");
 }
```

}

OUTPUT:

```
C:\java>javac MultipleCatches.java

C:\java>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

(II) Finally

Case -I : Exception NOT occurred

```
   class Finally_Case1
 {
public static void main(String args[])
{
try
{
int data=25/25;
System.out.println(data);
}
finally
{
System.out.println("finally block is  always executed");
}

System.out.println("rest of the code...");
}
}
```

OUTPUT:

```
C:\java>javac Finally_Case1.java

C:\java>java Finally_Case1
1
finally block is  always executed
rest of the code...
```

Case -2: Exception occurred and NOT handled

```
class Finally_Case2
{
public static void main(String args[])
{
try
{
int data=25/0;
System.out.println(data);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println(e);
}
finally
{
System.out.println("finally block is always executed");
}
 System.out.println("rest of the code...");
}
}
```

OUTPUT:

```
C:\java>javac Finally_Case2.java

C:\java>java Finally_Case2
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Finally_Case2.main(Finally_Case2.java:7)
```

Case – 3 : Exception occurred and handled

```
class Finally_Case3
{
public static void main(String args[])
{
try
{
int data=25/0;
 System.out.println(data);
}

catch(ArithmeticException e)
{

System.out.println(e);
}
finally
{
System.out.println("finally block is always executed");
}
 System.out.println("rest of the code...");
}
}
```

OUTPUT:

```
C:\java>javac Finally_Case3.java

C:\java>java Finally_Case3
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

(III) Creating user defined

```
class MyException extends Exception
{
String s ;
MyException( String s)
{
this.s=s;
}
public String toString()
{
return ("User Defined Exception" +s) ;
}
}
class UDefException
{
public static void main(String args[])
{
try
{
throw new MyException("MY EXP");
// throw is used to create a new exception and throw it.
}
catch(MyException e)
{
System.out.println(e) ;
}
}
```

}

OUTPUT:


C:\java>javac UDefException.java
C:\java>java UDefException
User Defined ExceptionMY EXP

EXERCISE-7

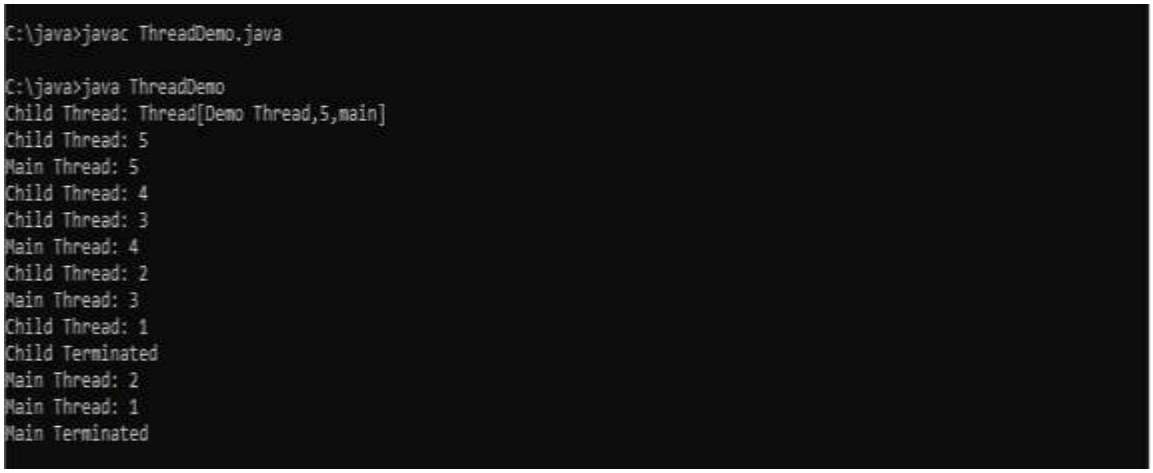
(a) Implement a program for Thread creation

(I) using Thread class.

```
class NewThread extends Thread
        {

         NewThread()
         {
          super("Demo Thread");
          System.out.println("Child Thread:"+this);
          start();
         }
         public void run()
         {
          try
          {
           for(int i=5;i>0;i--)
           {
            System.out.println("Child Thread:"+i);
                Thread.sleep(500);
           }
          }
          catch(InterruptedException e)
          {
             System.out.println("Child Interrupted");
          }
          System.out.println("Child Terminated!!");
         }
        }
        class ThreadDemo
        {
         public static void main(String args[])
         {
          new NewThread();
          try
          {
           for(int i=5;i>0;i--)
           {
            System.out.println("Main Thread:"+i);
                Thread.sleep(1000);
           }
          }
          catch(InterruptedException e)
          {
```

```
            System.out.println("Main Thread Interrupted");
          }
          System.out.println("Main Thread Terminated");
        }
      }
```

OUTPUT:



(II) using Runnable Interfaces.

```
class NewThread implements Runnable
  {
 String name;
 Thread t;
 NewThread(String n)
 {
  name=n;
  t=new Thread(this,name);
  System.out.println("Child Thread:"+t);
  t.start();
 }
 public void run()
 {
  try
  {
```

```
     for(int i=5;i>0;i--)
     {
       System.out.println(name+":"+i);
            Thread.sleep(500);
     }
    }
    catch(InterruptedException e)
    {
       System.out.println(name+"Interrupted");
    }
     System.out.println(name+"Terminated!!");
  }
}
class RThreadTest
{
  public static void main(String args[])
  {
    new NewThread("Thread1");

    try
    {
       for(int i=5;i>0;i--)
     {
        System.out.println("Main Thread"+":"+i);
       Thread.sleep(1000);
      }
    }
    catch(InterruptedException e)
    {
```

```
    System.out.println("Main Thread Interrupted");
   }
   System.out.println("Main Thread Terminated");
  }
}
```

OUTPUT:

(b) Implement a program for creating multiple threads.

```
class NewThread implements Runnable
  {
 String name;
 Thread t;
 NewThread(String n)
 {
  name=n;
  t=new Thread(this,name);
  System.out.println("Child Thread:"+t);
  t.start();
 }
 public void run()
 {
  try
  {
   for(int i=5;i>0;i--)
   {
    System.out.println(name+":"+i);
        Thread.sleep(500);
   }
  }
  catch(InterruptedException e)
```

```
    {
       System.out.println(name+"Interrupted");
    }
     System.out.println(name+"Terminated!!");
  }
}
class MThreadTest
{
  public static void main(String args[])
  {
    new NewThread("One");
    new NewThread("Two");
    new NewThread("Three");
    try
    {

          Thread.sleep(10000);
    }
    catch(InterruptedException e)
    {
      System.out.println("Main Thread Interrupted");
    }
    System.out.println("Main Thread Terminated");
   }
}
```

OUTPUT:

```
C:\java>javac MThreadTest.java

C:\java>java MThreadTest
Child Thread:Thread[One,5,main]
Child Thread:Thread[Two,5,main]
Child Thread:Thread[Three,5,main]
Three:5
One:5
Two:5
Three:4
One:4
Two:4
One:3
Three:3
Two:3
Two:2
One:2
Three:2
Two:1
One:1
Three:1
OneTerminated!!
TwoTerminated!!
ThreeTerminated!!
```

(c) Implement a program for Synchronization.

```
class Callme
{
   synchronized void call(String msg)
   {
      System.out.print("["+msg);
      try
      {
         Thread.sleep(1000);
      }
      catch(InterruptedException e)
      {
         System.out.println("Interrupted");
      }
      System.out.print("]");
   }
}
class Caller implements Runnable
{
   String msg;
   Callme target;
   Thread t;
   public Caller(Callme cm,String s)
   {
      target=cm;

   msg=s;
      t=new Thread(this);
      t.start();
   }
   public void run()
   {
      target.call(msg);
   }
}
class SynTest
{
   public static void main(String args[])
   {
      Callme cmob=new Callme();
      Caller ob1=new Caller(cmob,"hello");
      Caller ob2=new Caller(cmob,"Synchronized");
      Caller ob3=new Caller(cmob,"World");
      try
      {
         ob1.t.join();
         ob2.t.join();
```

```
            ob3.t.join();
          }
          catch(InterruptedException e)
          {
            System.out.println("Interrupted");
          }

        }
      }
```

OUTPUT:

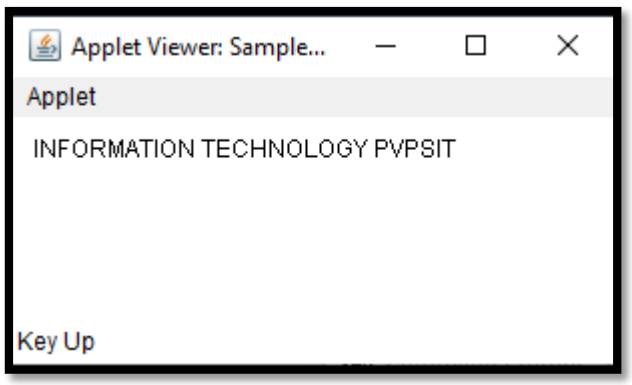
EXERCISE-8

a) Write a java program to demonstrate key Events .

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SampleKey" width=300 height=100>
</applet>
*/
public class SampleKey  extends  Applet  implements  KeyListener
{
String msg = "";
public void init()
{
addKeyListener(this);
}
public void keyPressed(KeyEvent ke)
```

```
{
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke)
{
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke)
{
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g)
{
g.drawString(msg, 10, 20);
}
}
```

OUTPUT:



(b) Write a Java program for handling mouse events.

```
import java.io.*;
import java.applet.Applet;
```

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

/*<applet code="MouseEx " width=500 height=100></applet>*/


public class MouseEx  extends Applet implements MouseListener,
                                                 MouseMotionListener
{
String txt=" ";
int x=10,y=30;

  public void init()
{
addMouseListener(this);
addMouseMotionListener(this);
}
public void mouseClicked(MouseEvent me)
{
txt="MOUSE CLICKED";
repaint();
}
public void mouseEntered(MouseEvent me)
{
txt="MOUSE ENTERED";
repaint();
}
public void mouseExited(MouseEvent me)
{
txt="MOUSE EXITED";
setForeground(Color.red);
repaint();
}
public void mousePressed(MouseEvent me)
{
txt="MOUSE PRESSED";
setForeground(Color.red);
repaint();
}
public void mouseMoved(MouseEvent me)
{
txt="MOUSE MOVED";
repaint();
}
public void mouseDragged(MouseEvent me)
{
txt="MOUSE DRAGGED";
```

```
repaint();
}
public void mouseReleased(MouseEvent me)
{
txt="MOUSE RELEASED";
repaint();
}
public void paint(Graphics g)
{
g.drawString(txt,20,40);
showStatus("Mouse event handling");
}
}
```

OUTPUT:

EXERCISE-9

a) Develop applications by using AWT controls

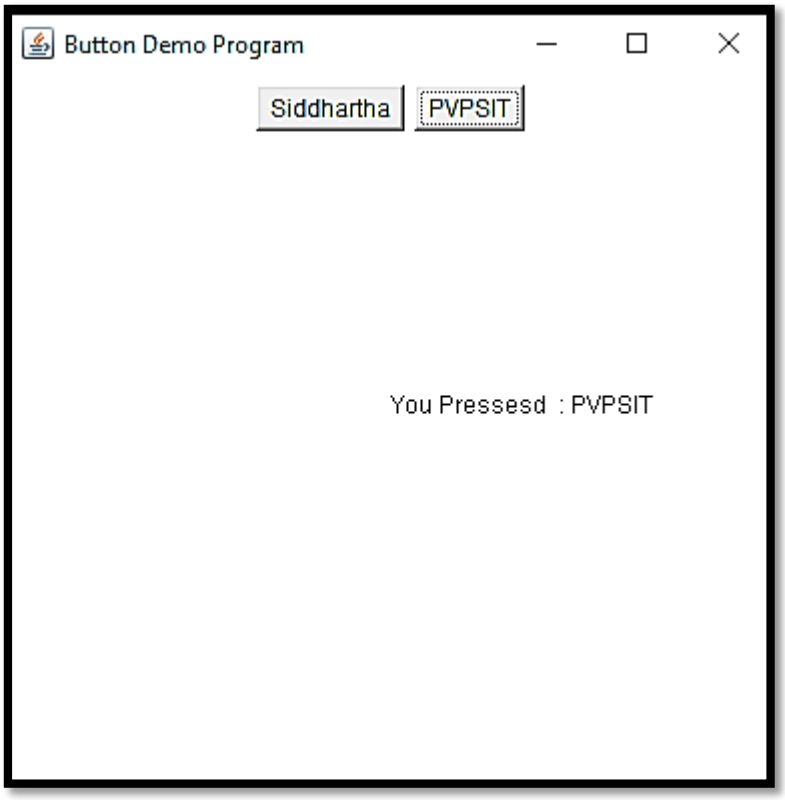
a. Buttons

Sol :

```
import  java.awt.*;
import  java.awt.event.*;
class  ButtonEx extends Frame implements  ActionListener
{
Button b1;
Button b2;
String msg="";
ButtonEx()
{
b1=new Button();
b2=new Button("PVPSIT");
b1.setLabel("Siddhartha");
add(b1);
add(b2);
b1.addActionListener(this);
b2.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
```

```
{
msg="You Pressesd  : "+ae.getActionCommand();
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg,200,200);
}
}
class AWTControlsDemo
{
public static void main(String arg[])
{
ButtonEx  d= new ButtonEx();
d.setSize(400,400);
d.setVisible(true);
d.setTitle("Button Demo Program");
d.setLayout(new FlowLayout());

d.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we)
{
System.exit(0);
}
});
}
}
```

OUTPUT:



b. TextField & TextArea

```
 import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet implements TextListener
{
TextField name, pass;
TextArea text;
String val = "This is TextArea";
public void init()
{
Label namep = new Label("Name: " );
name = new TextField(12);
Label passp = new Label("Password: " );
pass = new TextField(8);
pass.setEchoChar('*');
add(namep);
add(name);
```

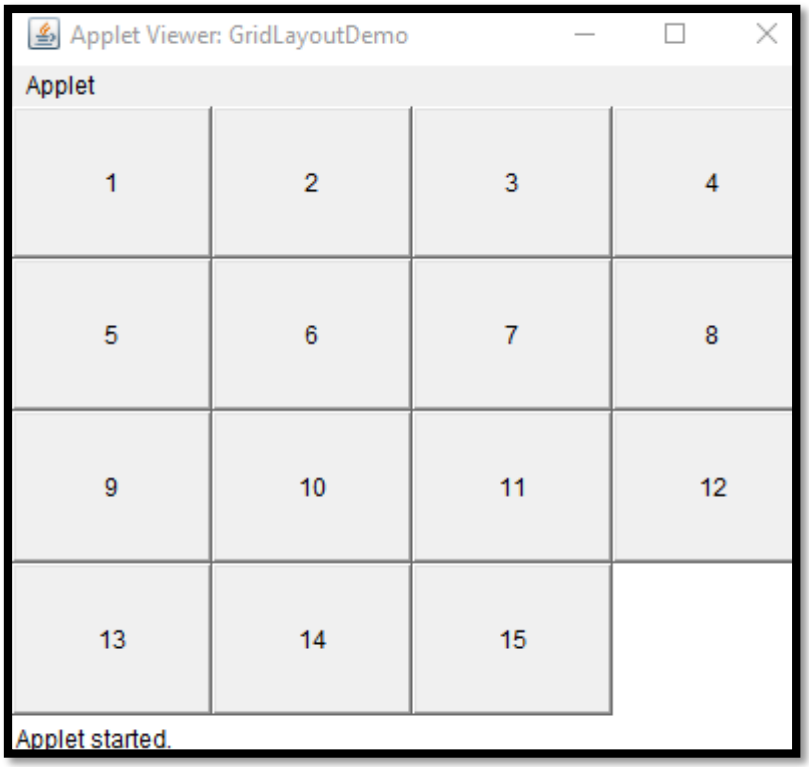
```
add(passp);
add(pass);
 text = new TextArea(val, 10, 15);
add(text);
//register to receive action events
name.addTextListener(this);
pass.addTextListener(this);
}
// User pressed Enter.
public void textValueChanged(TextEvent ae)
{
repaint();
}
public void paint(Graphics g)

{
g.drawString("Name Field : " + name.getText(), 6, 60);
g.drawString("Password Field: " + pass.getText(), 6, 100);
text.setText(name.getSelectedText());
}
}
```

OUTPUT:

c. GridLayoutManager

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

/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet
{
static final int n = 4;
public void init() {
setLayout(new GridLayout(n, n));
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));
}

}

}

}
```

OUTPUT:


Applet Viewer: GridLayoutDemo
Applet
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Applet started.

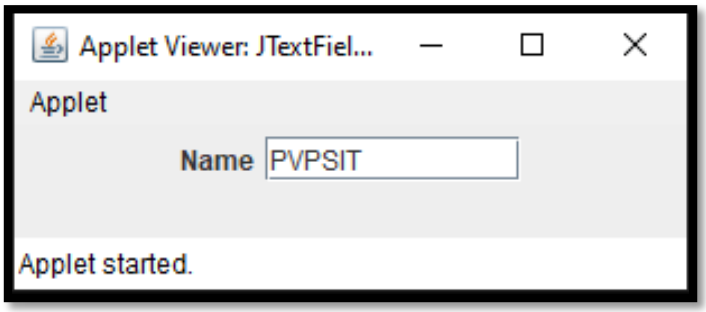
EXERCISE-10

Develop applications by using Swing componets

a. JLabel b. JTextField

```
import java.awt.*;
import javax.swing.*;
/*<applet code= "JTextFieldDemo" width=300 height=50>
</applet>*/
public class JTextFieldDemo extends JApplet
{
JLabel l;
JTextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
//Add Label , text field to content pane
l = new JLabel();
l.setText("Name");
jtf = new JTextField(10);
contentPane.add(l);
contentPane.add(jtf);
}
}
```

OUTPUT:

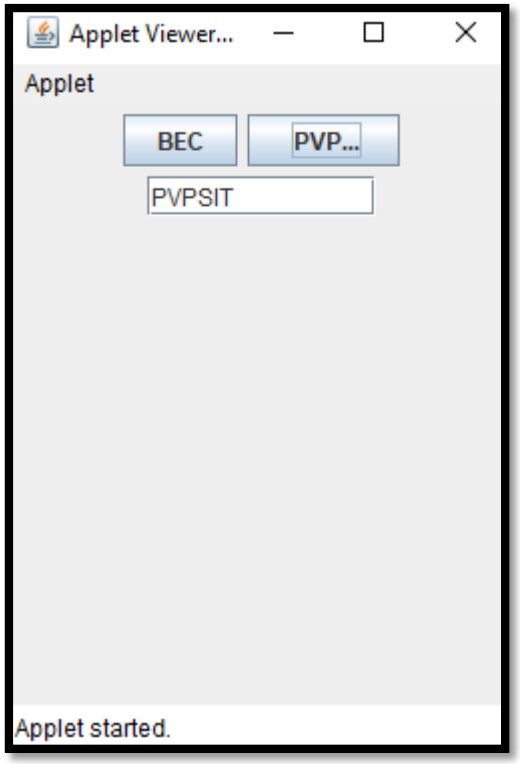


c. Develop application by using JButton

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
</applet>
*/
public class JButtonDemo extends JApplet implements ActionListener
{
JTextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane(); contentPane.setLayout(new FlowLayout());
JButton jb1 = new JButton("BEC"); jb1.addActionListener(this); contentPane.add(jb1);
// Add buttons to content pane
ImageIcon pvp = new ImageIcon("pvp.jpg");
JButton jb2 = new JButton("PVPSIT",pvp);
jb2.setActionCommand("PVPSIT");
jb2.addActionListener(this);
contentPane.add(jb2);
```

```
jtf = new JTextField(10);
contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae)
{
jtf.setText(ae.getActionCommand());
}
}
```

OUTPUT:

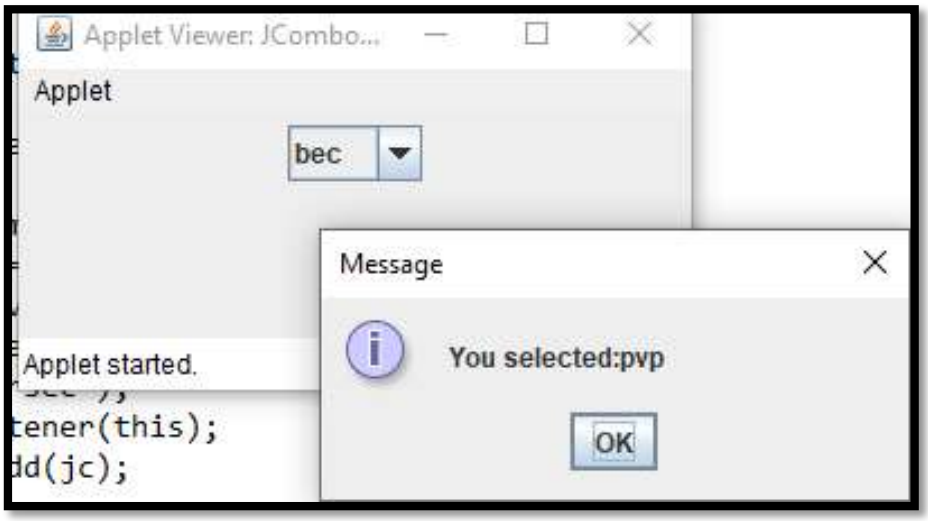


d. Develop application by using JComboBox.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/* <applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
```

```
public class JComboBoxDemo extends JApplet implements ItemListener
{
Container contentPane;
public void init()
{
// Get content pane
contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());

//Create a combo box and add it to the panel
JComboBox jc = new JComboBox();
jc.addItem("pvp");
jc.addItem("bec");
jc.addItem("vrsec");
jc.addItemListener(this);
contentPane.add(jc);
}
public void itemStateChanged(ItemEvent ie)
{
String s = (String)ie.getItem();
JOptionPane.showMessageDialog(null,"You selected:"+s);
}
}
```

OUTPUT:


Applet Viewer: JCombo...
Applet
bec
Applet started.
Message
You selected:pvp
OK
tener(this);
dd(jc);