OBJECT ORIENTED PROGRAMMING THROUGH JAVA

UNIT-I

Principles of OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- **Object**
- **Class**
- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Encapsulation**

1. Object: Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

The state of an object is stored in fields (variables), while methods (functions) display the object's behavior. Objects are created from templates known as classes. In Java, an object is created using the keyword "new".

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

2. Class

Collection of objects is called class. It is a logical entity. It represents the set of properties or methods that are common to all objects of one type

A class can also be defined as a blueprint from which objects are created. Class doesn't consume any space.

3. Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. 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).

It provides code reusability. It is used to achieve runtime polymorphism.

4.Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms. Polymorphism allows us to perform a single action in different ways.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

For example: To speak something; for example, a cat speaks meow, dog barks woof, etc.

Ex: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations.

5. Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

Another example of Abstraction is ATM Machine; All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement…etc. but we can't know internal details about ATM

Real Life Example of Abstraction

In Java, we use abstract class and interface to achieve abstraction.

6. Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from being accessed by other classes, thereby we can keep variables and methods safes from outside interference and misuse.

For example, a capsule, it is wrapped with different medicines.

When you log into your email accounts such as Gmail, Yahoo mail, or Rediff mail, there is a lot of internal processes taking place in the backend and you have no control over it.
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Applications of OOP

There are many applications of object oriented programming language,
Some of them are :

1. Real Time Systems Design.
2. Simulation and Modeling System.
3. Object Oriented Database.
4. Client-Server System.
5. Hypertext, Hypermedia.
6. Neural Networking and Parallel Programming.
7. CAD/CAM Systems.
8. AI and Expert Systems.

Java Variables:

Variable is name of *reserved area allocated in memory*. The value of a variable can change its value in the execution time. Java variables are classified into ***three*** types:

1. Local Variables: A variable which is declared inside the method is called local variable.

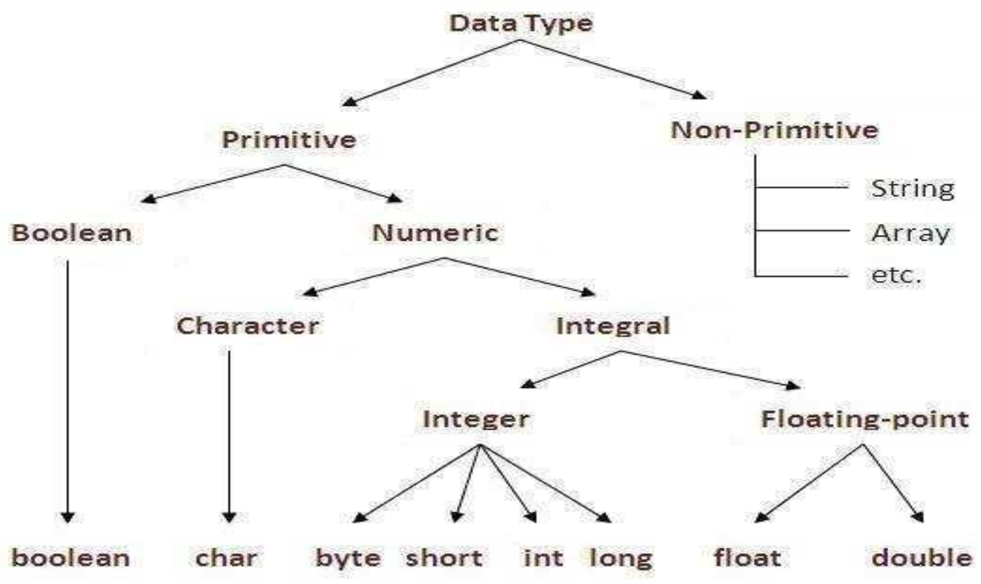
2. Class Variables (Static Variables): Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.

3. Instance Variables (Non-static Variables): A variable which is declared inside the class but outside the method, is called instance variable. It is not declared as static.

Primitive Data types: Types of data types:

1. Primitive data types
2. Non-primitive data types

Data Type	Default Value	Default size
boolean	FALSE	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
double	0.0d	8 byte



Identifiers -Naming Coventions:

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

In Java, there are several points to remember about identifiers. They are as follows −

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

- Examples of legal identifiers: age, $salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Keywords:

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract	Assert	Boolean	break	Byte	case	Catch	char
Class	Const	Continue	default	Do	double	Else	enum
extends	Final	Finally	float	For	goto	If	implements
Import	Instance of	Int	interface	Long	native	new	package
Private	Protected	Public	return	Short	static	Strict fp	super
Switch	synchronized	This	throw	Throws	transient	Try	void
Volatile	While						

Literals:

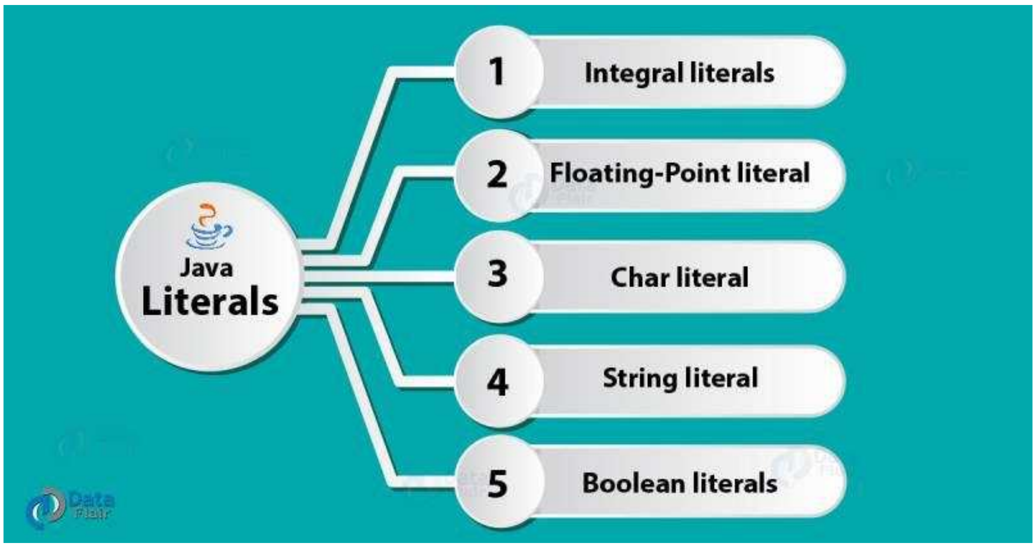
Literals are number, text, or anything that represent a value. In other words, Literals in Java are the constant values assigned to the variable. It is also called a ***constant****.*

For example, int x = 100;

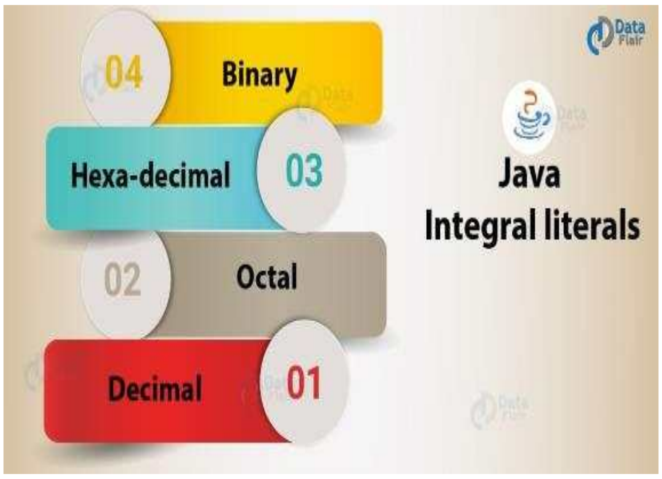
So, 100 is literal.

There are 5 types of Literals can be seen in Java..

Types of Literals in Java



1. Integral Literals in Java: We can specify the integer literals in 4 different ways

-

Decimal (Base 10) Digits from 0-9 are allowed in this form.

Ex: Int x = 101;

Octal (Base 8) Digits from 0 – 7 are allowed. It should always have a prefix 'o'.

Ex: int x = o146;

Hexa-Decimal (Base 16) Digits 0-9 are allowed and also characters from ***a-f*** are allowed in this form.

Ex: int x = OX123Face;

Binary A literal in this type should have a prefix ob and oB, from 1.7 one can also specify in binary literals, i.e. 0 and 1.

Ex: int x = ob1111;

2. Floating-Point Literals in Java

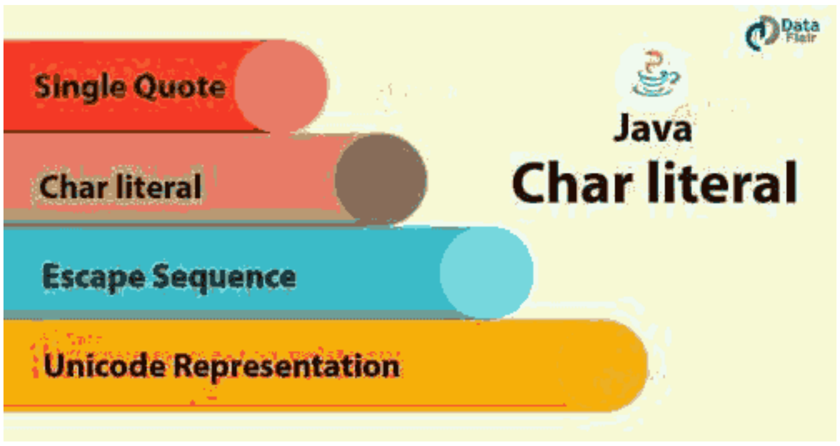
Floating point literals provide values that can be used where you need a float or double instance. There are three kinds of floating point literal.

The floating point literal consists of one or more decimal digits and a decimal point '.' and an optional suffix (f, F, d, D). The optional suffix allows you to specify that the literal is a float (f or F) or double (d or D) value. The default (when no suffix is specified) is Double.

For example : 0.0,1.0F,1.0D,3.14159

3. Char Literals in Java

These are the four types of char-



Single Quote: Java Literal can be specified to a char data type as a single character within a single quote.

Ex: char ch = 'a';

Char as Integral: A char literal in Java can specify as integral literal which also represents the Unicode value of a character.
Furthermore, an integer can specify in decimal, octal and even hexadecimal type, but the range is 0-65535.

Ex: char ch = 062;

Unicode Representation: Char literals can specify in Unicode representation '\uxxxx'. Here XXXX represents 4 hexadecimal numbers.

Ex: char ch = '\u0061';// Here /u0061 represent a.

Escape Sequence: Escape sequences can also specify as char literal.

Ex: char ch = '\n';

4. String Literals: Java String literals are any sequence of characters with a double quote.

Ex: String s = "Hello";

5. Boolean Literals : They allow only two values i.e. true and false.

Ex: boolean b = true;

Operators-Binary, Unary and ternary

Basic Operators in Java:

Java provides a rich set of operators to manipulate variables. The operators are classified into following groups:

1. Unary Operators
2. Arithmetic Operators

3. Relational Operators
4. Logical Operators
5. Assignment Operators
6. Bitwise Operators
7. Ternary Operators

1. **Unary Operator**

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Ex:

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
11
12
11
10
```

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

+ (Addition)	Adds values on either side of the operator	A + B will give 30
-(Subtraction)	Subtracts right-hand operand from left-hand operand	A - B will give -10
* (Multiplication)	Multiplies operands on either side of operator	A * B will give 200
/ (Division)	Divides left hand operand by right hand operand	B / A will give 2
% (Modulus)	Divides left operand by right and returns remainder	B % A will give 0

++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

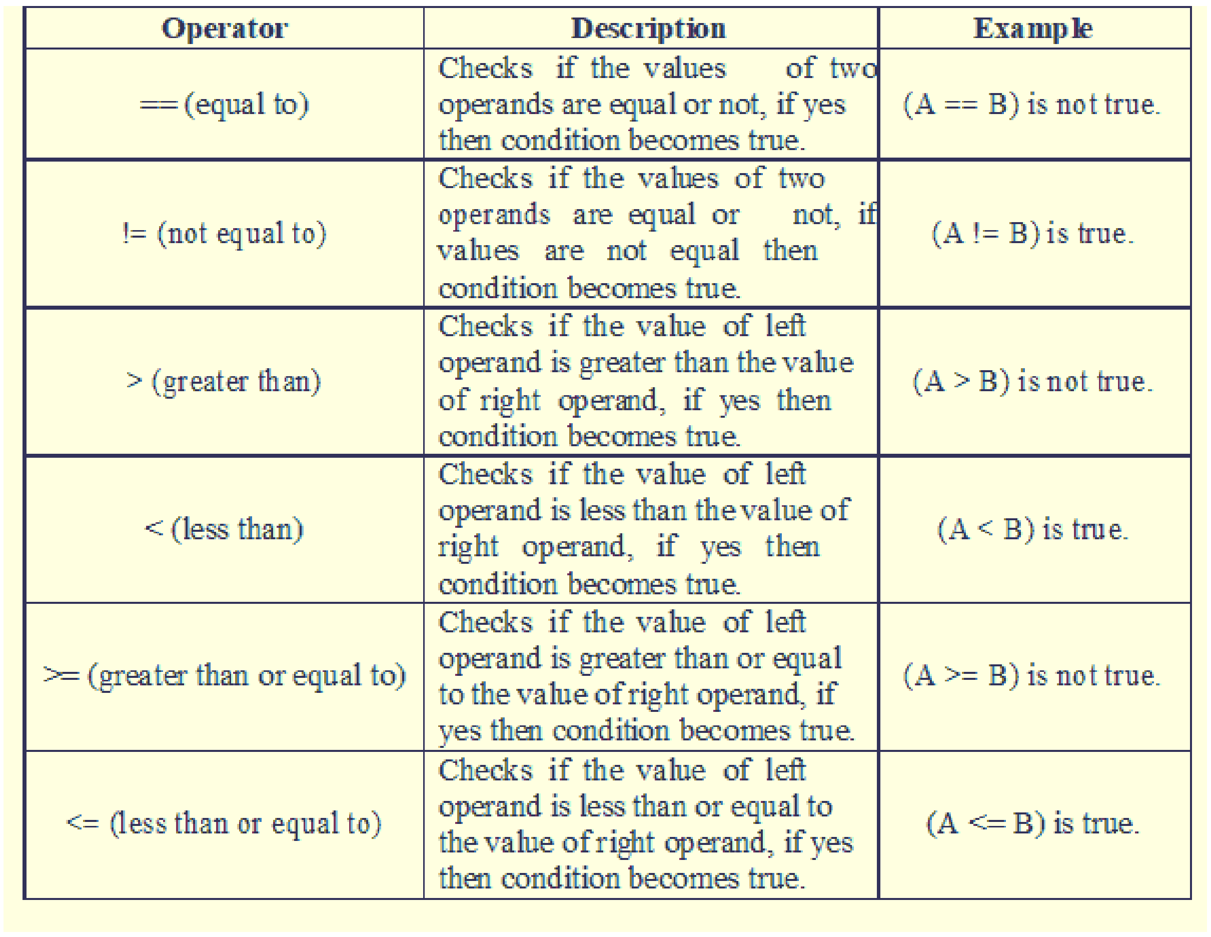
Ex:

```
      public class Arithops {
           public static void main(String args[]) {
                          int a = 10;
                          int b = 20;
                          int c = 25;
                          int d = 25;
                          System.out.println("a + b = " + (a + b) );
                          System.out.println("a - b = " + (a - b) );
                          System.out.println("a * b = " + (a * b) );
                          System.out.println("b / a = " + (b / a) );
                          System.out.println("b % a = " + (b % a) );
                          System.out.println("c % a = " + (c % a) );
   }
}
```

Output:

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
--    = 11
d++   = 25
++d   = 27
```

3.Relational Operators: There are following relational operators supported by Java language. Assume variable A holds 10 and variable B holds 20, then:



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

Ex:

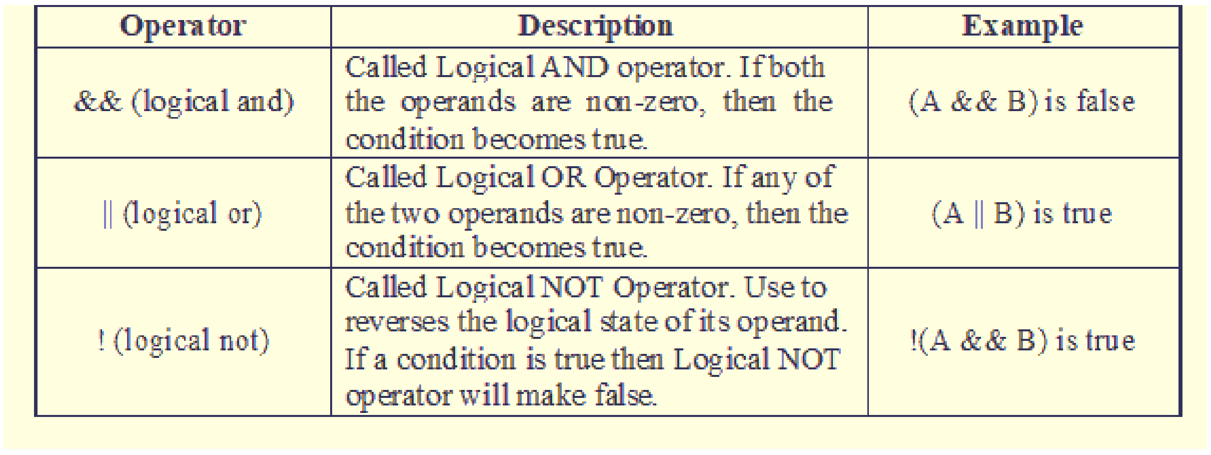
```
public class relops {
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
  }
}
```

Output:

```
a == b = false
```

```
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

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



Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

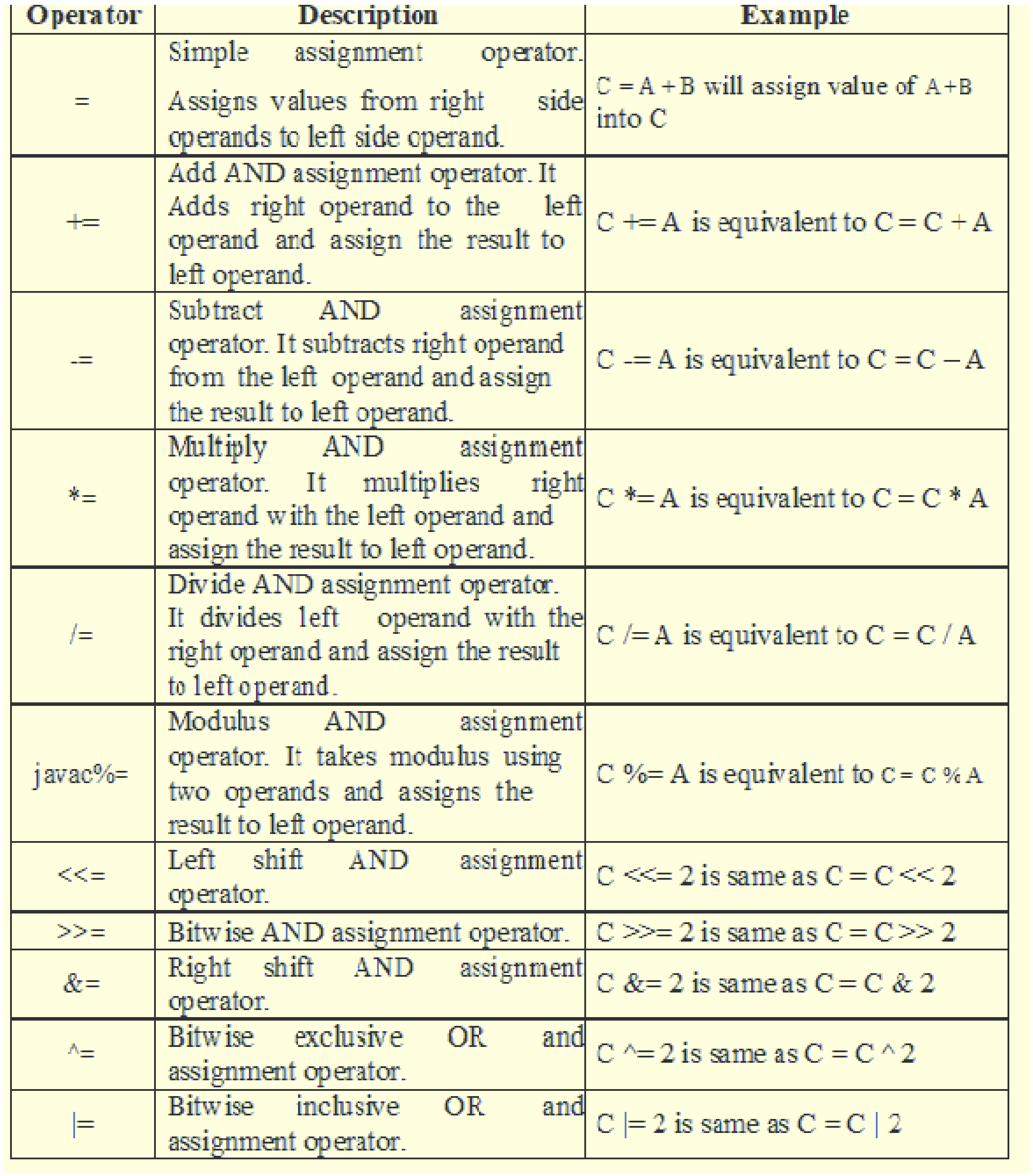
Ex:

```
public class logicalops {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    System.out.println("a && b = " + (a&&b));
    System.out.println("a || b = " + (a||b) );
    System.out.println("!(a && b) = " + !(a && b));
  }  }
```

Output:

```
a && b = false
a || b = true
!(a && b) = true
```

5.Assignment Operators: The following table lists the assignment operators supported by the Java language.



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

Ex:

```
class assignops{
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 0;
    c = a + b;
```

```
        System.out.println("c = a + b = " + c );
        c += a ;
        System.out.println("c += a  = " + c );
        c -= a ;
        System.out.println("c -= a = " + c );
        c *= a ;
        System.out.println("c *= a = " + c );
        a = 10;
        c = 15;
        c /= a ;
        System.out.println("c /= a = " + c );
        a = 10;
        c = 15;
        c %= a ;
        System.out.println("c %= a  = " + c );
        c <<= 2 ;
        System.out.println("c <<= 2 = " + c );
        c >>= 2 ;
        System.out.println("c >>= 2 = " + c );
        c >>= 2 ;
        System.out.println("c >>= 2 = " + c );
        c &= a ;
        System.out.println("c &= a  = " + c );
        c ^= a ;
        System.out.println("c ^= a   = " + c );
        c |= a ;
        System.out.println("c |= a   = " + c );
      }
    }
```

Output:

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
```

c /= a = 1

c %= a = 5

c <<= 2 = 20

c >>= 2 = 5

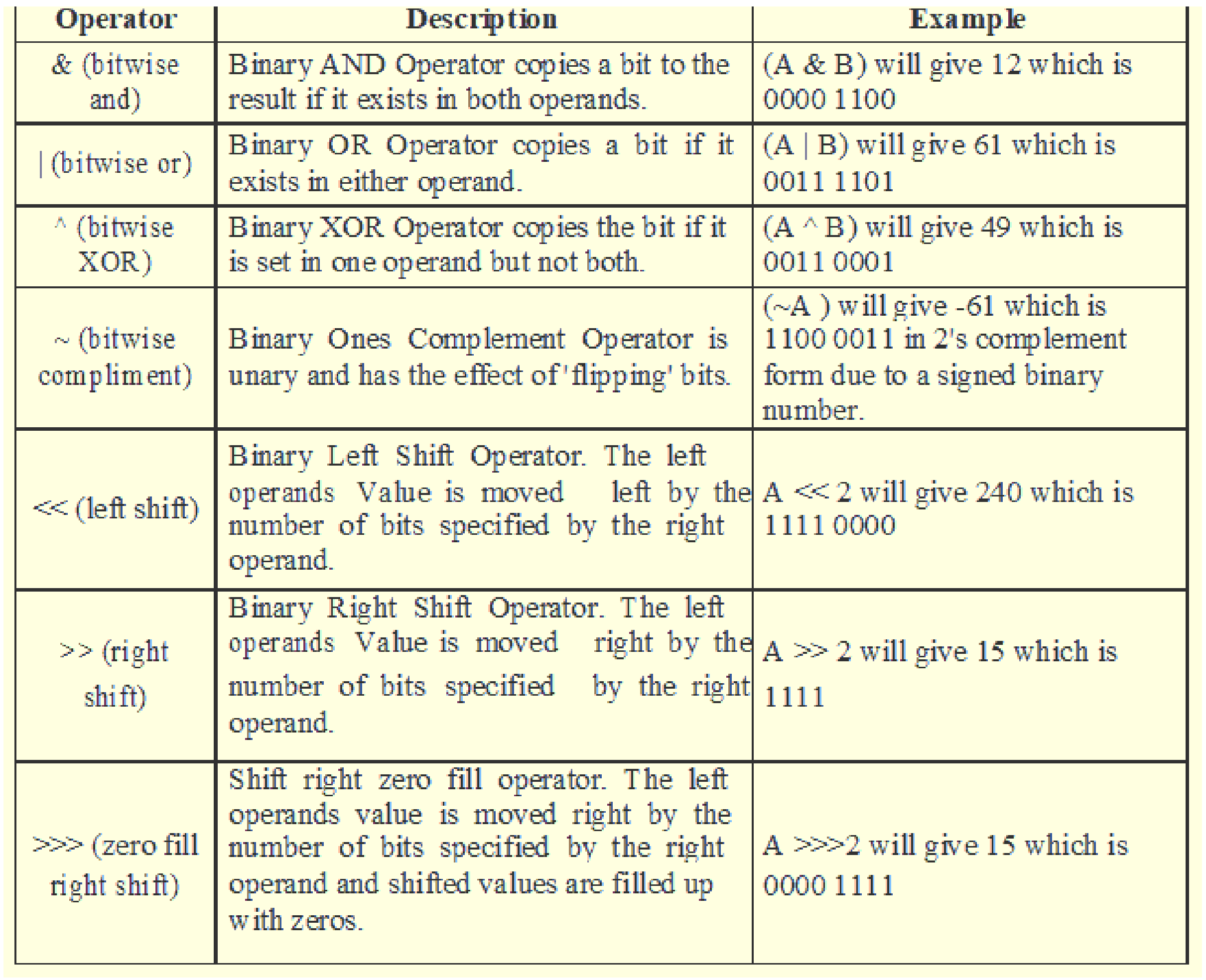
c >>= 2 = 1

c &= a = 0

c ^= a   = 10

c |= a   = 10

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



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

a = 0011 1100

b = 0000 1101

----

a & b = 0000 1100

a | b = 0011 1101

a ^ b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators. Assume integer variable A holds 60 and variable B holds 13 then:

Ex:

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

Output:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

7. Ternary Operator / Conditional Operator (? :)

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

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

Ex:

```
public class condop {
  public static void main(String args[]) {
     int a, b;
     a = 10;
     b = (a == 1) ? 20: 30;

     System.out.println( "Value of b is : " +  b );
     b = (a == 10) ? 20: 30;
     System.out.println( "Value of b is : " + b );
  }
}
```

Output:

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

Expressions:

Expressions consist of variables, operators, literals and method calls that evaluates to a single value.

Ex: int score;

Score=90;

Here, score = 90 is an expression that returns int.

Ex : Double a = 2.2, b = 3.4, result;

result = a + b - 3.4;

Here, a + b - 3.4 is an expression.

Ex:

```
if (number1 == number2)
System.out.println("Number 1 is larger than number 2");
```

Here, number1 == number2 is an expression that returns Boolean. Similarly, "Number 1 is larger than number 2" is a string expression.

Precedence rules and Associativity:

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

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3 * 2 and then adds into 7.

In the following table, operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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

Logical AND	➢ &&	Left to right
Logical OR	➢ ||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left

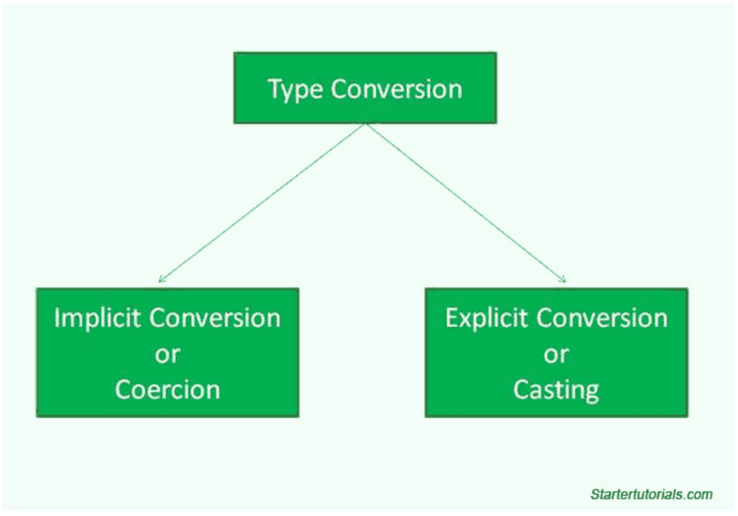
Primitive Type Conversion and Casting:

Converting a value from one type to another type (data type) is known as type conversion.

Type conversion is of two types based on how the conversion is performed:

1) ***Implicit conversion*** (also known as automatic conversion or coercion),

2) ***Explicit conversion*** (also known as type casting).


Type Conversion
Implicit Conversion or Coercion
Explicit Conversion or Casting
Startertutorials.com

Implicit Conversion or Coercion

This type of conversion is performed automatically by Java due to performance reasons. Implicit conversion is not performed at all times.

There are two rules to be satisfied for the conversion to take place. They are:

1. The source and destination types must be compatible with each other.
2. The size of the destination type must be larger than the source type.

For example, Java will automatically convert a value of *byte* into *int* type in expressions since they are both compatible and *int* is larger than *byte* type.

Since a smaller range type is converted into a larger range type this conversion is also known as **widening conversion**. Characters can never be converted to *boolean* type. Both are incompatible.

Explicit Conversion or Casting

There may be situations where you want to convert a value having a type of size less than the destination type size. In such cases Java will not help you. You have to do it on your own explicitly. That is why this type of conversion is known as explicit conversion or casting as the programmer does this manually.

Syntax for type casting is as shown below:

(destination-type) value

An example for type casting is shown below:

int a = 10;

byte b = (int) a;

In the above example, I am forcing an integer value to be converted into a *byte* type. For type casting to be carried out both the source and destination types must be compatible with each other. For example, you can't convert an integer to boolean even if you force it.

In the above example, size of source type *int* is 32 bits and size of destination type *byte* is 8 bits. Since we are converting a source type having larger size into a destination type having less size, such conversion is known as **narrowing conversion.**

A type cast can have unexpected behavior. For example, if a *double* is converted into an *int*, the fraction component will be lost.

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  byte -> short -> char -> int -> long -> float -> double
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char -> short -> byte

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class MyClass {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double
    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

Output:

```
9
9.0
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

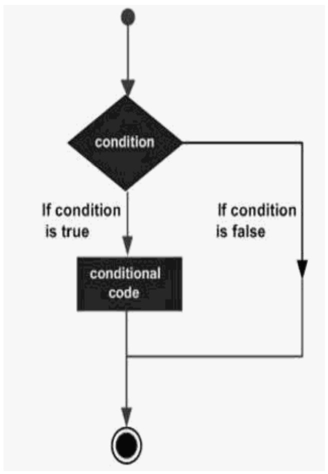
Example

```
public class MyClass {
  public static void main(String[] args) {
    double myDouble = 9.78;
    int myInt = (int) myDouble; // Manual casting: double to int
    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

Flow of control- Branching, Conditional, loops:

Decision Making and Branching Structures:

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. The general form of decision making structure is described in the following diagram:



Java programming language provides following types of decision making statements:

1. if statement
2. switch statement
3. conditional operator (? :)

1. if statement: if statement is having *FOUR* types of forms. They are

a. Simple if statement

b. if .. else statement

c. Nested if statement

d. else if Ladder

a. Simple if statement: An **if statement** consists of a boolean expression followed by one or more statements. The general format is

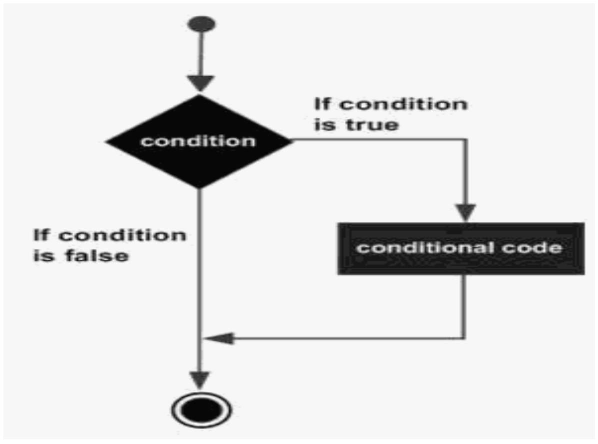
if(Boolean_expression) {

// Statements will execute if the Boolean expression is true

}

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

Flow Diagram:



Ex:

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

Output:

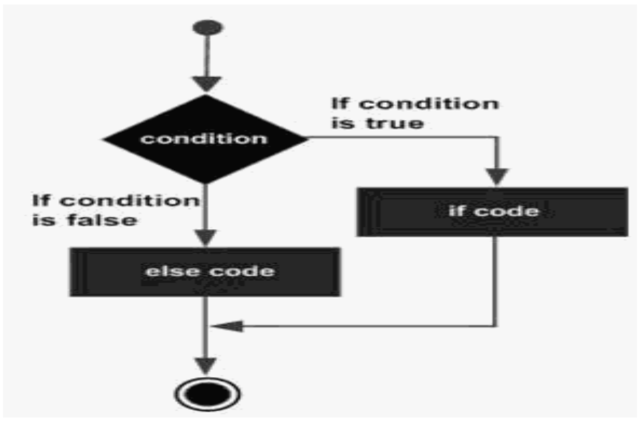
```
This is if statement
```

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

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

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram:



Ex:

```
public class Test {
  public static void main(String args[]) {
    int x = 30;
      if( x < 20 ) {
```

```
                System.out.print("This is if statement");
            }
        else {
            System.out.print("This is else statement");
            }
          }
        }
```

Output:

```
        This is else statement
```

c. Nested if statement: It is always legal to nest if-else statements which means we can use one if or else if statement inside another if or else if statement. The general format is

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

Ex:

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

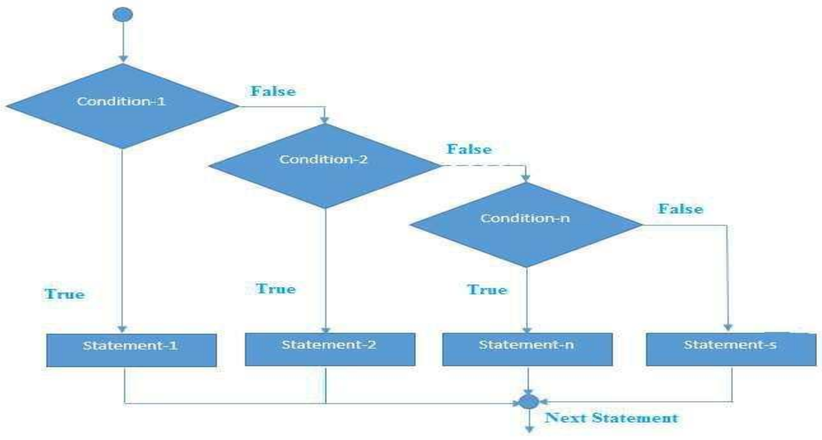
Output:

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

d. else .. if Ladder: The if-else-if ladder statement executes one condition from multiple statements. The general format is

```
if(condition1){
        //code to be executed if condition1 is
true }else if(condition2){
        //code to be executed if condition2 is true
}
else if(condition3){
        //code to be executed if condition3 is true
}
...
else{
        //code to be executed if all the conditions are false
}
```

Flow Diagram:



Ex:

```
class elseifladder {
        public static void main(String[] args) {
                int marks=65;
              if(marks<50){
                        System.out.println("fail");
                        }
                        else if(marks>=50 && marks<60){
                                System.out.println("D grade");


                        }
                        else if(marks>=60 && marks<70){
                                System.out.println("C grade");
                        }
                        else if(marks>=70 && marks<80){
```

```
                System.out.println("B grade");
            }
            else if(marks>=80 && marks<90){
                System.out.println("A grade");
            }else if(marks>=90 && marks<100){
                System.out.println("A+ grade");
            }else{
                System.out.println("Invalid!");
            }
        }
    }
```

Output:

```
    C grade
```

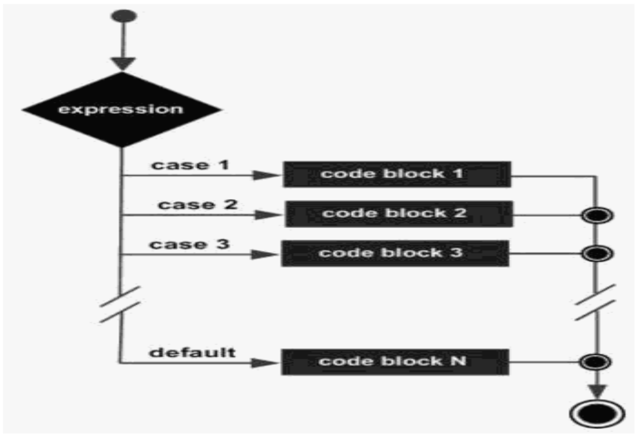
2. switch statement: A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. The general format is

```
switch(expression) {
  case value1 :
    // Statements
    break; //
    optional
   case value2 :
    // Statements
    break; //
    optional
          :
  case valueN :
    // Statements
    break; //
    optional
   default : // Optional
    // Statements
}
```

The following rules apply to a **switch** statement:

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

Flow Diagram:



Ex:

```
class switch {
  public static void main(String args[]) {
    // char grade = args[0].charAt(0);
    char grade = 'C';
    switch(grade) {
```

```
                    case 'A' :
                      System.out.println("Excellent!");
                      break;
                    case 'B' :
                      System.out.println("Good!");
                      break;
                    case 'C' :
                      System.out.println("Well done");
                      break;
                    case 'D' :
                      System.out.println("You passed");
                      break;
                    case 'F' :
                      System.out.println("Better try again");
                      break;
                    default :
                      System.out.println("Invalid grade");
                  }
                  System.out.println("Your grade is " + grade);
                }
              }
```

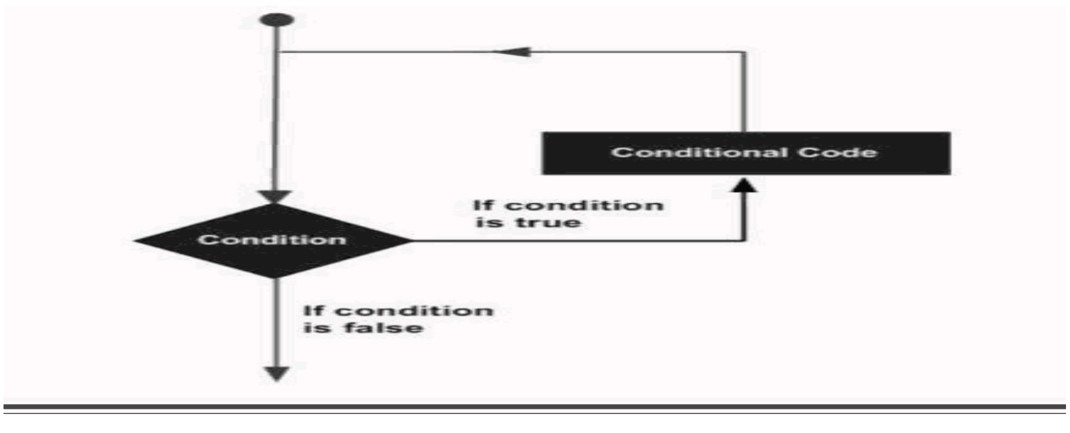
Output:

```
              Well done
              Your grade is C
```

3. conditional operator (? :): ***This is covered in the operators concept***.

Loop Statements:

A **loop** statement allows us to execute a statement or group of statements multiple times. The general format is



Java provides the following THREE types of loop statements:

1. while loop
2. do .. while loop
3. for loop

1. while loop: A **while** loop statement repeatedly executes a target statement(s) as long as a given condition is true. The general format is

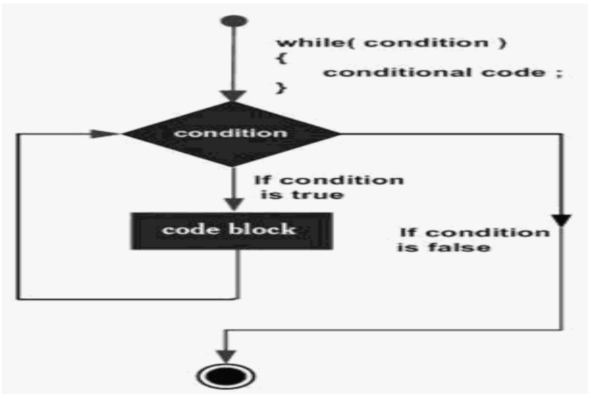
while(Boolean_expression) {

// Statements

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

Flow Diagram:



Ex:

```
class whileloop {
  public static void main(String args[]) {
    int x = 10;
    while( x < 16 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }  }
```

Output:

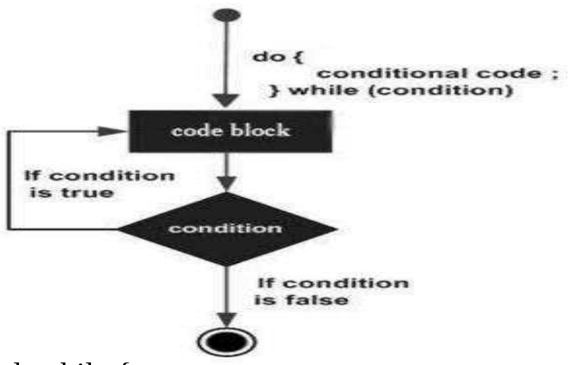
```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
```

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

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

The Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested. If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Flow Diagram:



Ex:

```
class dowhile {
  public static void main(String args[]) {
    int x = 10;
    do {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 16 );
  }
}
```

Output:

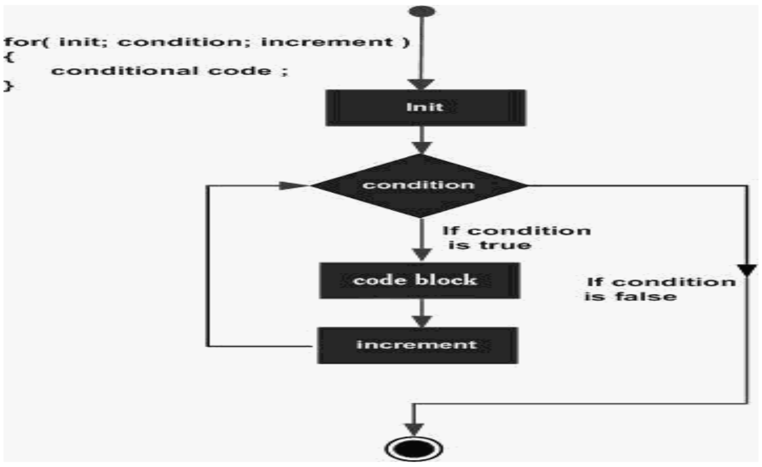
```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
```

3. for loop: A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A **for** loop is useful when we know how many times a task is to be repeated. The general format is

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

a. The **initialization** step is executed first, and only once. This step allows us to declare and initialize any loop control variables and this step ends with a semi colon (;).
b. Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement after the for loop block.
c. After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows us to update any loop control variables. This statement can be left blank with a semicolon at the end.
d. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram:



Ex:

```
public class forloop {
  public static void main(String args[]) {
    for(int x = 10; x < 16; x++) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
    }
  }
}
```

Output:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
```

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Java supports the following control statements.

1. break statement
2. continue statement

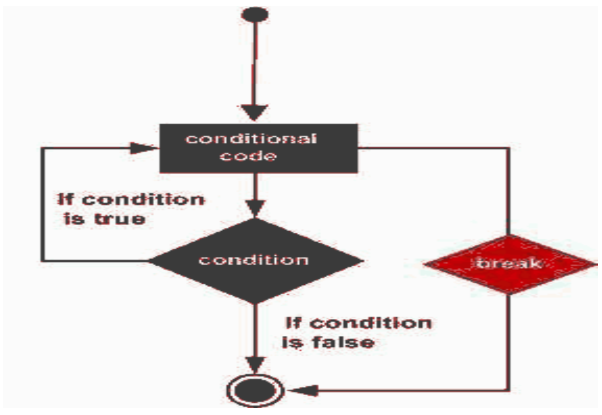
1. break statement: The **break** statement in Java programming language has the following two usages:

a. When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
b. It can be used to terminate a case in the **switch** statement.

The general format is:

break;

Flow Diagram:



Ex:

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

Output:

```
10
20
```

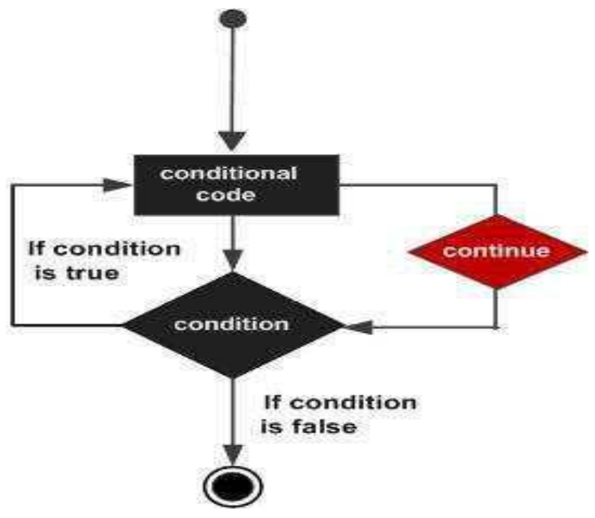
2. continue statement: The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

a. In a for loop, the continue keyword causes control to immediately jump to the update statement.
b. In a while loop or do/while loop, control immediately jumps to the Boolean expression.

The general format is:

continue;

Flow Diagram:



Ex:

```
class continuedemo {
  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x : numbers ) {
      if( x == 30 ) {
        continue;
      }
      System.out.print( x + "   ");
    }
      System.out.println( );
  }
}
```

Output:

```
10 20 40 50
```

Enhanced "for" loop in Java:

The enhanced for loop was introduced in ***Java 5***. This is mainly used to traverse collection of elements including arrays. The general format is:

for(declaration : expression) {

// Statements

}

a. **Declaration** − The newly declared block variable, is of a type compatible with the **e**lements of the array you are accessing. The variable will be available within the "for" lock and its value would be the same as the current array element.

b. **Expression** − This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Ex:

```
class continuedemo {
  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};
    for(int x : numbers ) {
      System.out.print( x );
      System.out.print(",");
```

```
        }
        System.out.print("\n");
        String [] names = {"James", "Larry", "Tom", Lacy"};
        for( String name : names ) {
           System.out.print( name );
           +System.out.print(",");
        }
      }  }
```

Output:

```
10, 20, 30, 40, 50,
James, Larry, Tom, Lacy,
```

Labelled Loops: In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, place it before the loop with a colon at the end. For example, a block of statements can be labelled as the following:

```
block1:  {
              ……………….
              ……………….
         block2: {
                    ……………..
                    ……………...
          }
         …………………..
         …………………..
  }
```

1. The ***break or continue*** statement can be followed by a label.
2. The presence of a label will transfer control to the start of the code identified by the label.

The general format is:

break label;

continue label;

Ex:

```
class breakcontinuelabel {
```

```
public static void main(String[] args) {
  OuterLoop: for (int i = 2;; i++) {
    for (int j = 2; j < i; j++) {
      if (i % j == 0) {
        continue OuterLoop;
      }
    }
    ++System+.out.print(i + " ");
    if (i == 37) {
      break OuterLoop;
    }
  }
 }
}
```

Output: 2 3 5 7 11 13 17 19 23 29 31 37

Cleaning up unused objects-Garbage collector:

Class variable and Methods-Static keyword, this keyword, Arrays

Command line arguments:

Classes and Objects

Class: *"A class is a collection of objects of similar type. Once a class is defined, any number of objects can be produced which belong to that class." A Class is a blueprint from which individual objects are created.* There is no Java program without a class.

The general format of a class is

```
class <clsname> {
        Variable declaration;
        Methods definition;
}
```

a. ***Class*** is a keyword which is used for creating user-defined data type.
b. ***Class name*** represents a JAVA valid variable name and it is treated as name of the class.
c. Class contains two parts namely ***variable declaration*** and ***methods definition***. ***Variable declaration*** represents what type of data members we can use of the class. ***Method definition*** represents to perform an operation.

Ex for Class:

```
public class Dog {
  String breed;
   int age;
   String color;
   void barking() {
   }
   void hungry() {
   }
   void sleeping() {
   }
}
```

Object: Objects have states and behaviors. If we consider the real-world, we can find many objects around us cars, dogs, humans, etc. All these objects have a **state** and a **behavior**.

If we consider a dog, then its **state** is - **name, breed, color** and the **behavior** is - **barking, wagging the tail, running**.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a **state** and a **behavior**. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the ***new*** keyword is used to create new objects.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

The general format for creating an object is

<clsname> objname = new <clsname ( )>;

a. **clsname** represents name of the class.
b. **Objname** represents JAVA valid variable name treated as object.

c. **new** is called dynamic memory allocation operator.

d. **clsname ()** represents constructor.

Following is an example of creating an object –

Example

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }
   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

Output: Passed Name is: Tommy

Accessing Class Members:

The general format to access the members of a class is

object_name.variable_name

object_name.method_name(parameters_list);

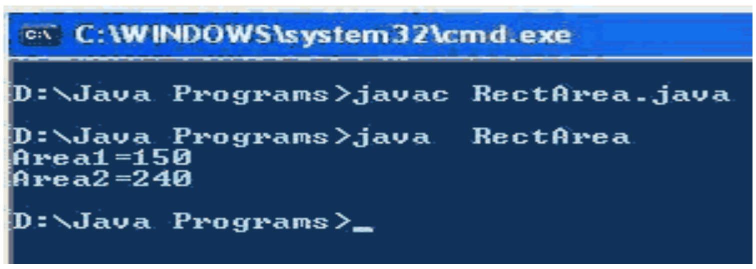
Here *object_name* is the name of the object, *variable_name* is the name of the instance variable inside the object that we need to access, *method_name* is the method that we wish to call and *parameter_list* is a comma separated list of *actual values (expressions)*.

Ex:

```
class Rectangle {
        int length,width;          // Declaration of variables
        void getData(int x, int y) {        //Declaration of method
                length=x;
                width=y;
        }
        int rectArea( ){              //Definition of another method
                int area=length*width;
                return (area);     }  }
```

```
class RectArea { // class with main method
        public static void main(String args[]) {
                int area1, area2;
                Rectangle r1=new Rectangle();       // Creating objects
                Rectangle r2=new Rectangle();
                r1.length=15;           //Accessing Variables
                r1.width=10;
                area1=r1.length * r1.width;
                r2.getData(20,12);      //Accessing methods
                area2=r2.rectArea( );
                System.out.println("Area1=" + area1);
                System.out.println("Area2=" + area2);
        } }
```

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Programs>javac RectArea.java

D:\Java Programs>java  RectArea
Area1=150
Area2=240

D:\Java Programs>_
```

Class Variables:

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Methods In Java: A **method** is a block of code which only runs when it is called.

You can pass data, known as **parameters**, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Methods are used -To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

Example

Create a method inside MyClass:

```
public class MyClass {
  static void myMethod() {
    // code to be executed
  }
}
```

- myMethod() is the name of the method
- static means that the method belongs to the MyClass class and not an object of the MyClass class.
- void means that this method does not have a return value.

Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, myMethod() is used to print some text, when it is called:

Example

Inside main, call the myMethod() method:

```
public class MyClass {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "I just got executed!"
```

A method can also be called multiple times:

Example

```
public class MyClass {
  static void myMethod() {
    System.out.println("I just got executed!");
  }
  public static void main(String[] args) {
    myMethod();
    myMethod();
    myMethod();
  }
}
// I just got executed!
// I just got executed!
// I just got executed!
```

Differences between Class and Object:

S. No.	Class	Object
1	A ***class*** is a way of binding the data and associated methods in a single unit.	Class variable is known as an ***Object***.
2	Whenever we start executing a java program, the class will be loaded into main memory with the help of class loader Sub system (part of JVM) only once.	After loading the class into main memory, objects can be created in ‘ ***n***’ number.
3	When a class is defined there is no memory space for data members of a class.	When an object is created we get the memory space for data members of the *class*.

Constructors:

A constructor is a special member method which will be called by the JVM implicitly (automatically) for placing user/programmer defined values instead of placing default values. Constructors are meant for initializing the object.

Advantages:

1. A constructor eliminates placing the default values.
2. A constructor eliminates calling the normal method implicitly.

Rules/Properties/Characteristics of a Constructor:

a. Constructor name must be similar to name of the class.

b. Constructor should not return any value even void also (if we write the return type for the constructor then that constructor will be treated as ordinary method).

c. Constructors should not be static since constructors will be called each and every time whenever an object is creating.

d. Constructor should not be private provided an object of one class is created in another class (constructor can be private provided an object of one class created in the same class).

e. Constructors will not be inherited at all.

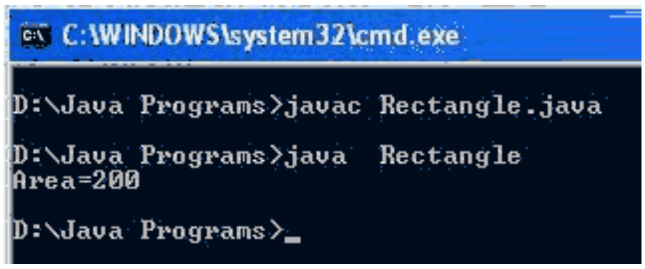
f. Constructors are called automatically whenever an object is creating.

Ex:

```
        class Rectangle {
                int length;
                int width;
                Rectangle(int x, int y) {
                        length=x;
                        width=y;
                }
                int area( ) {
                return (length*width);
                }
                public static void main(String[] args) {
//Constructor is implicitly called when we create an object
               Rectangle r1=new Rectangle(10,20); //calling constructor
               int x=r1.area();              //calling the method
               System.out.println("Area=" + x);
```

```
                }
        }
```

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Programs>javac Rectangle.java

D:\Java Programs>java  Rectangle
Area=200

D:\Java Programs>_
```

Types of Constructors:

Based on creating objects in JAVA we have two types of constructors. They are default/parameter less/no argument constructor and parameterized constructor.

Default Constructor:

This constructor will not take any parameters. Whenever we create an object only with default constructor, defining the default constructor is optional. If we are not defining default constructor of a class, then JVM will call automatically system defined default constructor (SDDC). If we define, JVM will call user/programmer defined default constructor (UDDC).

The general format is

```
class <clsname> {
        clsname ( ) {        //default constructor
                Block of statements;
                ........................;
                ........................;
        }
        ........................;
        ........................;
}
```

Parameterized Constructor:

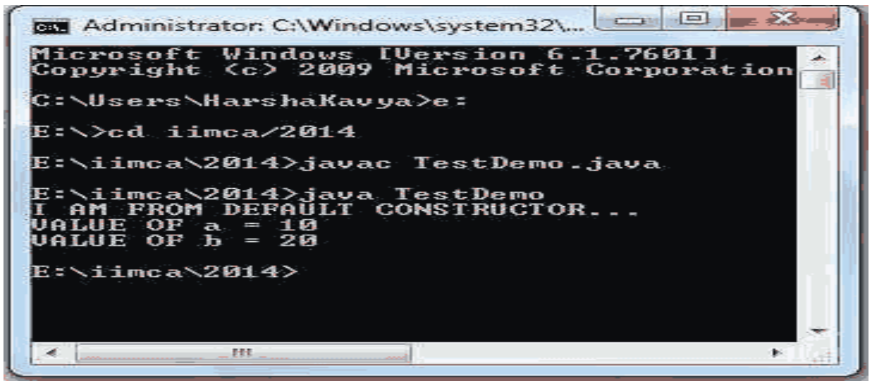
This constructor takes some parameters. Whenever we create an object using parameterized constructor, it is mandatory for the JAVA programmer to define parameterized constructor otherwise we will get compile time error. The general format is

```
class <clsname> {
        ………………………….;
        ………………………….;
        <clsname> (list of parameters) { //parameterized constructor
                Block of statements (s);
        }
        ………………………….;
        ………………………….;
}
```

Ex:

```
class Test {
    int a, b;
    Test () {
            System.out.println ("I AM FROM DEFAULT CONSTRUCTOR...");
            a=10;
            b=20;
            System.out.println ("VALUE OF a = "+a);
            System.out.println ("VALUE OF b = "+b);
    }
}
class TestDemo {
    public static void main (String [] args) {
            Test t1=new Test ();
    }
}
```

Output:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation

C:\Users\HarshaKavya>e:

E:\>cd iimca/2014

E:\iimca\2014>javac TestDemo.java

E:\iimca\2014>java TestDemo
I AM FROM DEFAULT CONSTRUCTOR...
VALUE OF a = 10
VALUE OF b = 20

E:\iimca\2014>
```

Note: 1

Whenever we define/create the objects with respect to both parameterized constructor and default constructor, it is mandatory for the JAVA programmer to define both the constructors.

Note: 2

Overloaded Constructor is one in which constructor name is similar but its signature is different. Signature represents number of parameters, type of parameters and order of parameters. Here, at least one thing must be differentiated. *For example:*

```
Test t1=new Test (10, 20);
Test t2=new Test (10, 20, 30);
Test t3=new Test (10.5, 20.5);
Test t4=new Test (10, 20.5);
Test t5=new Test (10.5, 20);
```

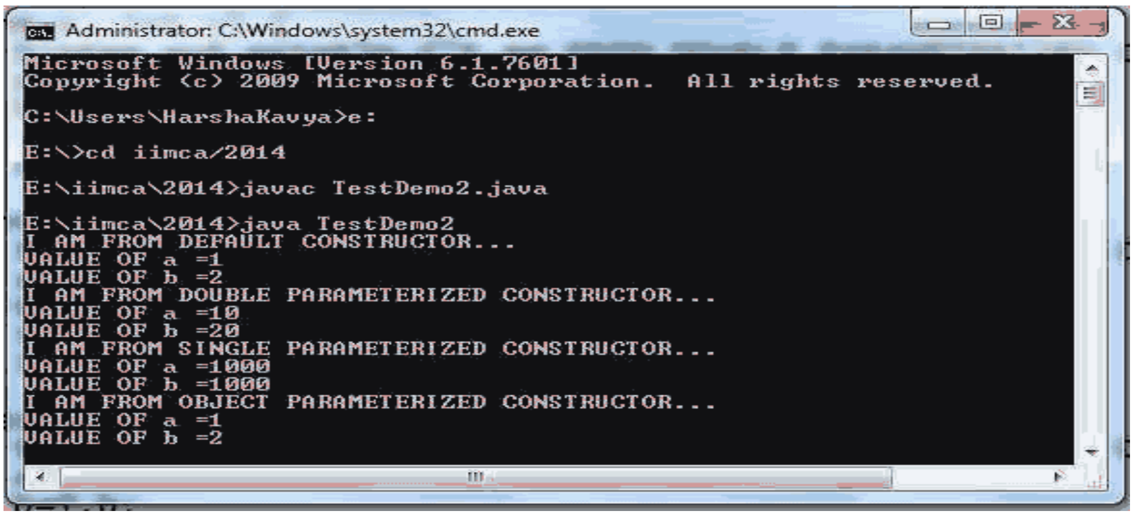
Note: 3

By default the parameter passing mechanism is call by reference.

Ex: *The following program which illustrates the concept of default constructor, parameterized constructor and overloaded constructor*.

```
class Test {
    int a, b;
    Test () {
        System.out.println ("I AM FROM DEFAULT CONSTRUCTOR...");
        a=1;
        b=2;
        System.out.println ("VALUE OF a ="+a);
        System.out.println ("VALUE OF b ="+b);
    }
    Test (int x, int y) {
        System.out.println ("I AM FROM DOUBLE PARAMETERIZED
CONSTRUCTOR...");
        a=x;
        b=y;
        System.out.println ("VALUE OF a ="+a);
        System.out.println ("VALUE OF b ="+b);
    }
    Test (int x) {
            System.out.println ("I AM FROM SINGLE PARAMETERIZED
CONSTRUCTOR...");
            a=x;
            b=x;
            System.out.println ("VALUE OF a ="+a);
            System.out.println ("VALUE OF b ="+b);
    }
    Test (Test T) {
            System.out.println ("I AM FROM OBJECT PARAMETERIZED
CONSTRUCTOR...");
            a=T.a;
            b=T.b;
            System.out.println ("VALUE OF a ="+a);
            System.out.println ("VALUE OF b ="+b);
    }
}
class TestDemo2 {
    public static void main (String args[]) {
            Test t1=new Test ();
            Test t2=new Test (10,20);
            Test t3=new Test (1000);
            Test t4=new Test (t1);
    }
```

Output:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\HarshaKavya>e:

E:\>cd iimca/2014

E:\iimca\2014>javac TestDemo2.java

E:\iimca\2014>java TestDemo2
I AM FROM DEFAULT CONSTRUCTOR...
VALUE OF a =1
VALUE OF b =2
I AM FROM DOUBLE PARAMETERIZED CONSTRUCTOR...
VALUE OF a =10
VALUE OF b =20
I AM FROM SINGLE PARAMETERIZED CONSTRUCTOR...
VALUE OF a =1000
VALUE OF b =1000
I AM FROM OBJECT PARAMETERIZED CONSTRUCTOR...
VALUE OF a =1
VALUE OF b =2
```

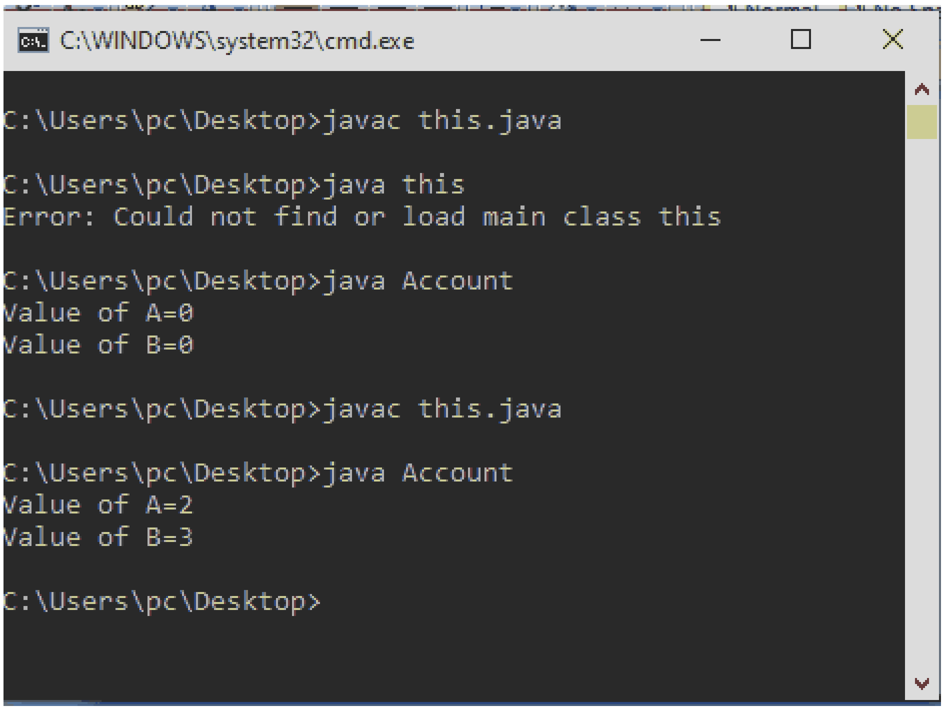
‘this ‘ Keyword: ‘this’ is an internal or implicit object created by JAVA for two purposes. They are

1. ‘this’ object is internally pointing to current class object.
2. Whenever the formal parameters and data members of the class are similar, to differentiate the data members of the class from formal parameters, the data members of class must be proceeded by ‘this’.

Ex: Demo on ‘this’ Keyword

```
class Account{
int a;
int b; //here a,b are instance variable
public void setData(int a,int b)    // here a,b are local variables
{
this.a=a;
this.b=b;
}
public void showData()
{
System.out.println("Value of A="+a);
System.out.println("Value of B="+b);
}
public static void main(String args[])
{
Account obj=new Account();
obj.setData(2,3);
obj.showData();
}
}
```

Output:



```
C:\Users\pc\Desktop>javac this.java

C:\Users\pc\Desktop>java this
Error: Could not find or load main class this

C:\Users\pc\Desktop>java Account
Value of A=0
Value of B=0

C:\Users\pc\Desktop>javac this.java

C:\Users\pc\Desktop>java Account
Value of A=2
Value of B=3

C:\Users\pc\Desktop>
```

Methods Overloading:

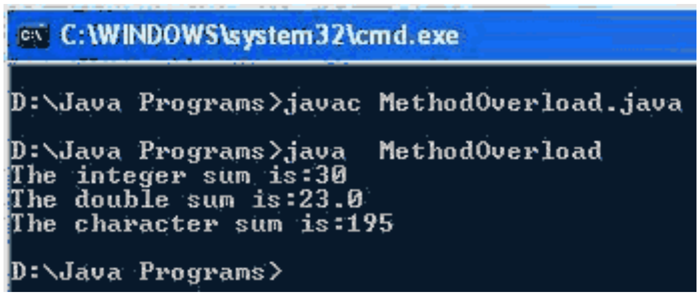
It is possible to create methods that have the same name, but different parameter list and different definitions in Java. This is called ***Method Overloading***. The following program illustrates the concept of method overloading in Java.

Ex:

```
class MethodOverload  {
    int sum(int x,int y) {
            int z;
            z=x+y;
            return z;
    }
    double sum(double x, double y) {
            double z;
            z=x+y;
            return z;
    }
    int sum(char x, char y) {
            int z;
            z=x+y;
            return z;
    }
```

```
        public static void main(String[] args) {
                MethodOverload o1=new MethodOverload();
                System.out.println("The integer sum is:" + o1.sum(10,20));
                System.out.println("The double sum is:" + o1.sum(10.5,12.5));
                System.out.println("The character sum is:" +o1.sum('a','b'));
    }
```

Output:



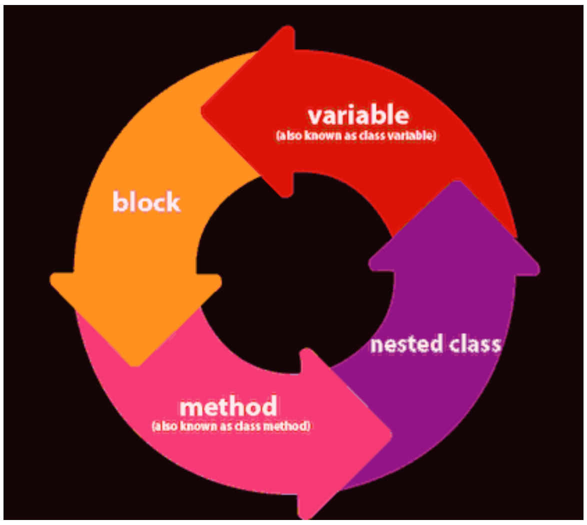
```
C:\WINDOWS\system32\cmd.exe

D:\Java Programs>javac MethodOverload.java

D:\Java Programs>java  MethodOverload
The integer sum is:30
The double sum is:23.0
The character sum is:195

D:\Java Programs>
```

Java static keyword

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){
System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 s1.display();
 s2.display();
 }
}
```

Output:

```
111 Karan ITS
```

```
222 Aryan ITS
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){
System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   //calling display method
   s1.display();
   s2.display();
  s3.display();
   }
}
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method

class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

Output:125

3) **Java static block**

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
  static{
 System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output:static block is invoked

```
Hello main
```

Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

4)Java static nested class example with instance method

```
class TestOuter1{
 static int data=30;
 static class Inner{
  void msg(){
  System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
```

Output: data is 30

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

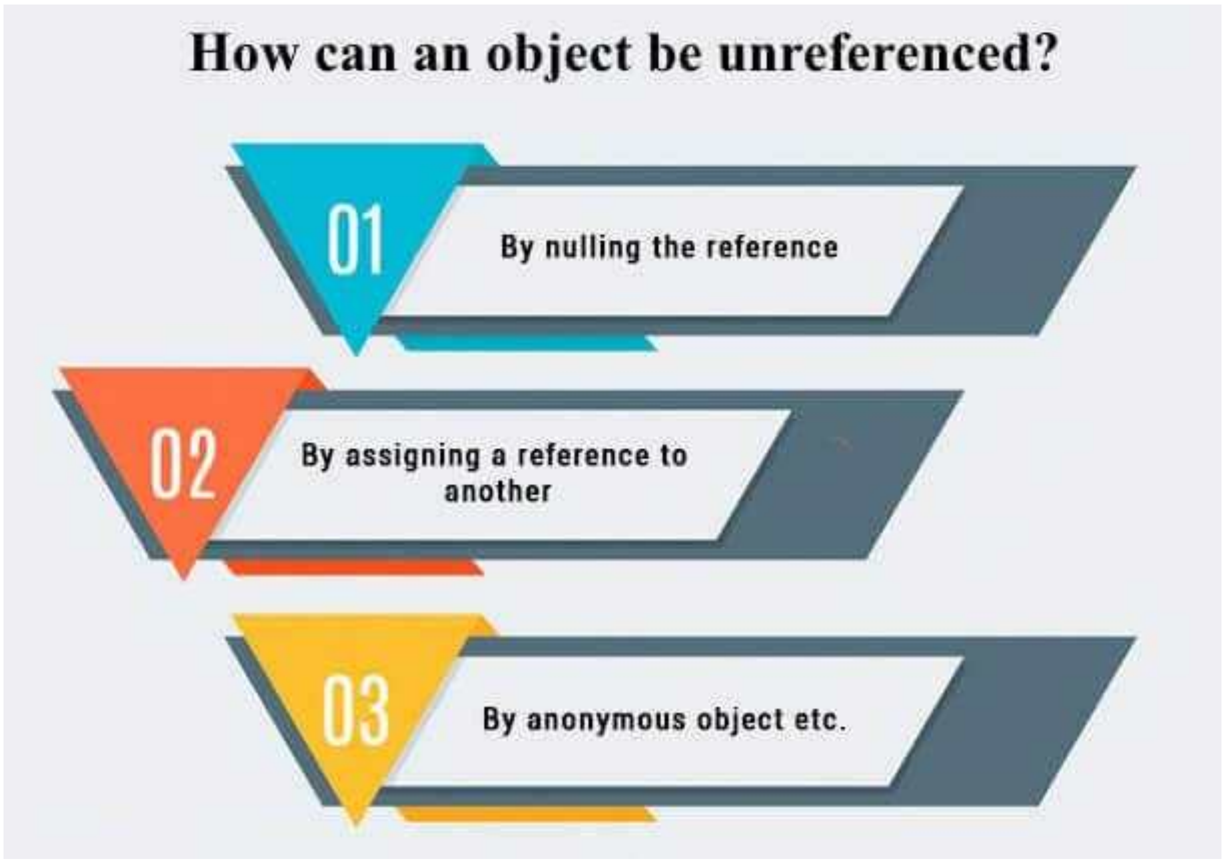
To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

An object be unreferenced in many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.


How can an object be unreferenced?
01
By nulling the reference
02
By assigning a reference to another
03
By anonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

---

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected void** finalize(){}

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

public static void gc(){}

Simple Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){
System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

Output:

```
object is garbage collected
object is garbage collected
```

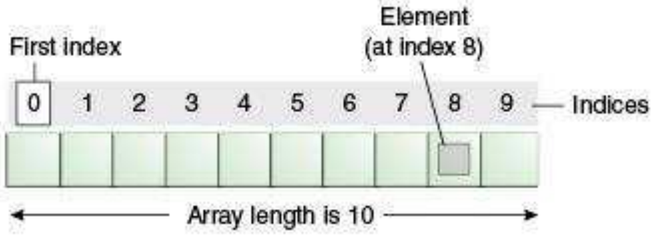
Arrays:

“ ***An array is a consecutive memory locations of same data type.***”

(or)

“ ***An Array is a derived data type which is a collection of similar data items that are to be stored in continuous memory locations and all the elements are referred by single name as array name.***”

Ex: An array of 10 elements is shown in the following diagram.



Advantages:

1) Array can store multiple values of same data type.
2) All the elements in array are accessible by using single name (array name). It overcomes the name conflict problem.
3) It is a capable of storing many elements at a time.
4) The memory locations of elements in the array are sequential.
5) Sorting and searching becomes easy.

Disadvantages:

1) Memory wastage will be there.
2) To delete an element in array. We need to traverse throughout an array.
3) The size of the array is fixed while declaration itself.
4) The size can’t be increased / decreased dynamically.

To create an array requires ***two*** steps. They are

a) Declare the array: Arrays in Java may be declared in two forms:

Form1: ***type array-name[];***

Form2: ***type[] array-name;***

Where, type refers data type of the array and array-name is any valid identifier.

Ex:
```
int number[];
float average[];
```

```
int[] counter;
```

b) Creation of arrays (Allocating the memory for an array):

After declaring an array, we need to create it in memory. Java allows us to create arrays using new operator only. The general format for allocating memory for an array is

array-name = new type[size];

Where, ***new*** is a keyword used to allocate memory for the array, and
s***ize*** is No of elements that the array will hold.
We can combine both the steps into a single one as shown below:

type array-name[] = new type[size];

Ex:
```
number=new int[5];
average=new float[10];
```

c) Initialization of Arrays: The final step is to put values into the array created. This process is known as ***initialization***. The general format is

array-name[subscript] = value;

(Or)

type array-name[]={list of values separated by commas};

***Ex:* 1**

```
int a[] = new int[3];
a[0] = 10;
a[1] = 20;
a[2] = 30;
```

***Ex:* 2**

```
int a[] = {1,2,3,4,5};
```

In the above example, 'a' is an integer type array with size automatically set to 5. All the 5 elements are initialized with the values 1, 2, 3, 4, 5. a[0] is 1, a[1] is 2, a[2] is 3, a[3] is 4 and a[4] is 5.

Array Length

In Java, all arrays store the allocated size in a variable named length. We can obtain the length of the array using the ***'length'*** .

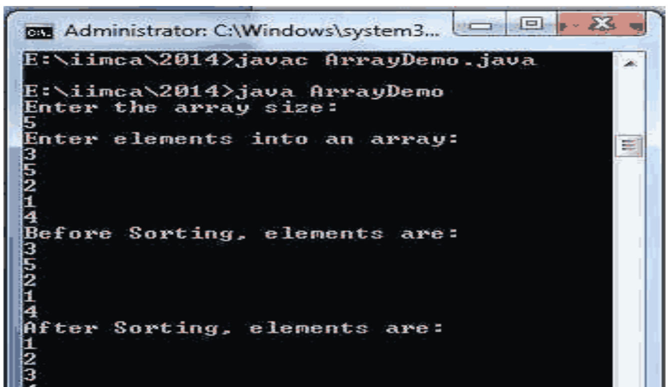
Ex:

```
int number[]={10,20,30,40,50};
int n=number.length; // We can get n=5
```

Ex: *To sort an array of elements in ascending order.*

```
import java.util.Scanner;
class ArrayDemo {
        public static void main(String[] args) {
                int a[];
                int n,temp;
                Scanner scan=new Scanner(System.in);
                System.out.println("Enter the array size:");
                n=scan.nextInt();
                a=new int[n];
                System.out.println("Enter elements into an rray:");
                for(int i=0;i<n;i++) {
                        a[i]=scan.nextInt();
                }
                System.out.println("Before Sorting element are:");
                for(int i=0;i<n;i++)
                System.out.println(a[i]);
                // Bubble Sort Logic
                for(int i=0;i<n-1;i++) {
                      for(int j=0;j<n-1-i;j++) {
                                        if(a[j]>=a[j+1]) {
                                                temp=a[j];
                                                a[j]=a[j+1];
                                                a[j+1]=temp;
                                         }
                        }
                }
                System.out.println("After Sorting, elements re:");
                for(int i=0;i<n;i++)
                        System.out.println(a[i]);
        }
}
```

Output:



```
E:\iimca\2014>javac ArrayDemo.java

E:\iimca\2014>java ArrayDemo
Enter the array size:
5
Enter elements into an array:
3
5
2
1
4
Before Sorting, elements are:
3
5
2
1
4
After Sorting, elements are:
1
2
3
```

Two-Dimensional Arrays:

Two dimensional array is nothing but array of array. An Array which is used to represent and store data in a tabular form is called Two Dimensional Array. Such types of arrays are used to represent data in a matrix form. A two dimensional array is declared as the following:

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

Where ‘ **a**’ is a two dimensional array which can hold 12 elements in 3 rows and 4 columns.

Initializing Two Dimensional Array:

a) int a[][] = new int[2][2];

a[0][0] = 1;

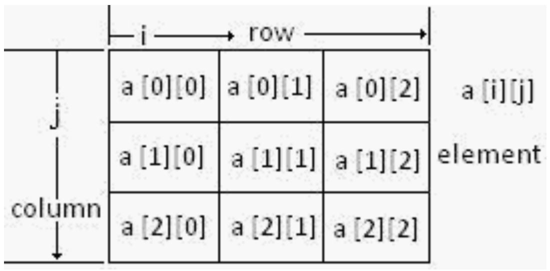
a[0][1] = 2;

a[1][0] = 3;

a[1][1] = 4;

b) int a[3][3];

Memory Allocation:



c) We can directly initialize the elements of a two dimensional array as shown below: int a[][] = { {1,2},{3,4}};

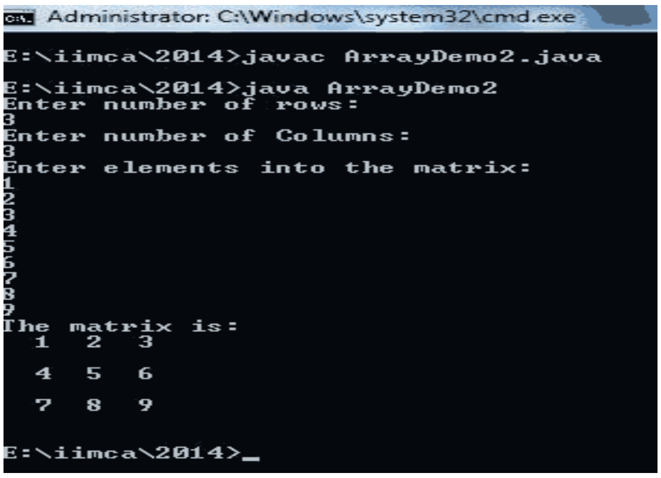
d) Alternative way of declaring a two dimensional array is:

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

The following program illustrates the concept of two-dimensional arrays in java.

```
import java.util.Scanner;
class ArrayDemo2 {
      public static void main(String[] args) {
                  int a[][];
                  int m,n,i,j;
                  Scanner scan=new Scanner(System.in);
                  System.out.println("Enter number of rows:");
                  m=scan.nextInt();
                  System.out.println("Enter number of Columns:");
                  n=scan.nextInt();
                  a=new int[m][n];
                  System.out.println("Enter elements into matrix:");
                   for(i=0;i<m;i++) {
                              for(j=0;j<n;j++) {
                                          a[i][j]=scan.nextInt();
                              }
                  }
                  System.out.println("The matrix is:");
                  for(i=0;i<m;i++) {
                              for(j=0;j<n;j++)
                                    System.out.print("  " + a[i][j]);
                       System.out.println("\n");
                  }
      }
}
```

Output:



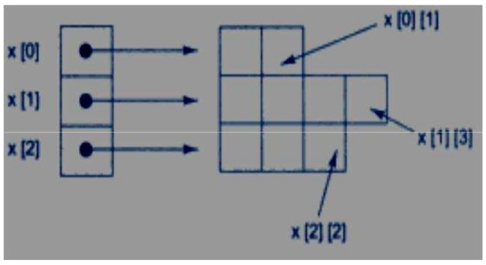
Administrator: C:\Windows\system32\cmd.exe

```
E:\iimca\2014>javac ArrayDemo2.java

E:\iimca\2014>java ArrayDemo2
Enter number of rows:
3
Enter number of Columns:
3
Enter elements into the matrix:
1
2
3
4
5
6
7
8
9
The matrix is:
  1  2  3

  4  5  6

  7  8  9

E:\iimca\2014>_
```

Variable Size Arrays: Java treats multidimensional arrays as " ***arrays of arrays***". It is possible to declare a two-dimensional array as follows.

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

These statements create a two-dimensional array as having different lengths for each row as shown in below.



COMMAND LINE ARGUMENTS:

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

Example

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

```
public class CommandLine {

  public static void main(String args[]) {

    for(int i = 0; i<args.length; i++) {

      System.out.println("args[" + i + "]: " + args[i]);

    }

  }

}
```

Try executing this program as shown here -

```
$java CommandLine 10 20 30 40 50
```

Output

This will produce the following result -

```
args[0]: 10
args[1]: 20
args[2]: 30
args[3]: 40
args[4]: 50
```

Inheritance: Extending a Class:

1. The process of obtaining the data members and member functions from one class to another class is known as ***Inheritance***.
2. The class which is giving data members and member functions to some other class is known as ***Base/ Super / Parent class***.
3. The class which is retrieving or obtaining the data members and member functions is known as ***Derived/Sub/Child class***.
4. A ***Derived class*** contains some of features of its own plus some of the data members from base class.

Advantages of Inheritance:

a. Application development time is less.
b. Application amount of memory space is less.
c. Redundancy (Repetition) of the code is reduced.

Defining a Subclass:

A subclass is defining as the following:

```
class <subclassname> extends <superclassname>
        { Variables declaration;
         Methods definitions;
}
```

The keyword ***'extends'*** is a keyword which is used for inheriting the data members and methods from base class to the derived class and it also improves functionality of derived class.

Types of Inheritance:

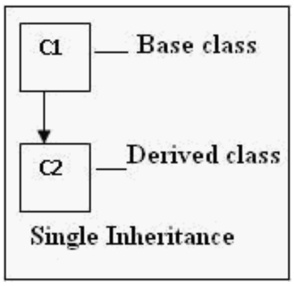
Java does not support Multiple Inheritance. But we can achieve it using interfaces.

Java Inheritance may take different forms such as

1. Single Inheritance (only one super class)
2. Multilevel Inheritance (several super classes)

3. Multiple Inheritance (several super class, many sub classes)
4. Hierarchical Inheritance (one super class, many sub classes)

1. Single Inheritance: It contains a single base class and a single derived class.

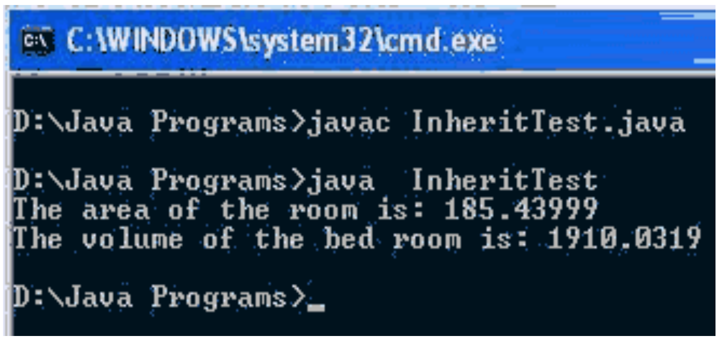


Ex:

```
class  Room  {
 float length, width;
 Room(float x, float y) {
         length=x;
         width=y;
 }
 float roomArea() {
         return (length * width);
 }  }
class BedRoom extends Room {
 float height;
 BedRoom(float x, float y, float z) {
         super(x,y);
         height=z;
 }
 float volume() {
         return (length * width * height);
 }
}
class InheritTest  {
 public static void main(String args[])  {
     BedRoom r1=new BedRoom(12.2f,15.2f,10.3f);
     float area=r1.roomArea();
```

```
        float vol=r1.volume();
          System.out.println("The area of the room is: " + area);
        System.out.println("The volume of the bed room is: " + vol);

    }
   }
```

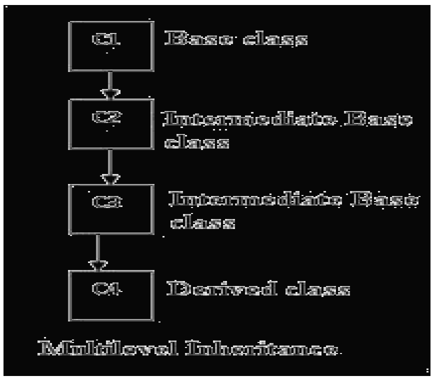
Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Programs>javac InheritTest.java

D:\Java Programs>java  InheritTest
The area of the room is: 185.43999
The volume of the bed room is: 1910.0319

D:\Java Programs>_
```

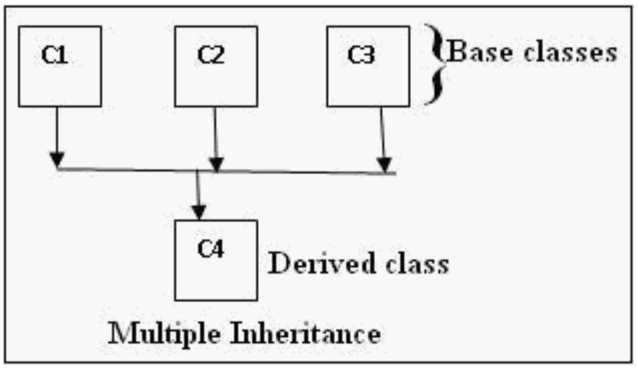
2. Multilevel Inheritance:

It consist one base class, one derived class and multiple intermediate base classes. An intermediate base class acts as a derived class, in another context the same class acts as a base class.

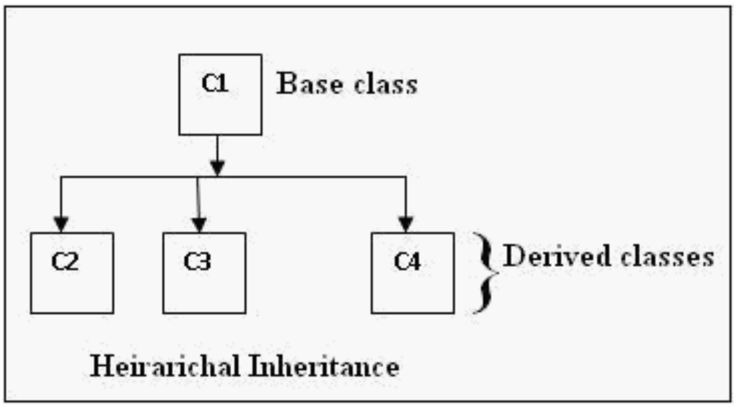


3. Multiple Inheritance:

It contains multiple base classes and a single derived class. The concept of multiple inheritance is not supported in java directly. But we achieve it using interfaces.



4. Hierarchical Inheritance: It consist single base class and multiple derived classes.



‘super’ Keyword in Java:

A Subclass constructor is used to construct the instance variables of both the subclass and the super class. The subclass constructor uses the keyword super to invoke the constructor method of the super class. The super keyword is used subject to the following conditions.

1. **‘super’** may only be used within a subclass constructor method.
2. The call to super class constructor must appear as the first statement within the subclass constructor.
3. The parameters in the ‘**super’** call must match the order and type of the instance variables declared in the super class.

Method Overriding:

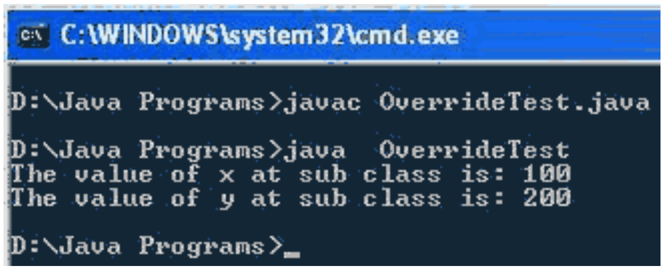
The process of redefining the same method for many number of implementations is called as method overriding.

The method overriding is possible by defining a method in the sub class that has the same name, same arguments and same return type as a method in the super class.

Ex:

```
class  Super {
  int x;
  Super(int x) {
          this.x=x;
  }
  void display() { // method defined
  System.out.println("The value of x at super is: " + x);
  }
}
class Sub extends Super {
  int y;
  Sub(int x, int y) {
          super(x);
          this.y=y;
  }
  void display() { //method defined again
    System.out.println("The value of x at sub class is: " + x);
    System.out.println("The value of y at sub class is: " + y);

  }
}
class OverrideTest {
  public static void main(String args[]) {
          Sub s1=new Sub(100,200);
          s1.display();
  }
}
```

Output:



C:\WINDOWS\system32\cmd.exe

```
D:\Java Programs>javac OverrideTest.java

D:\Java Programs>java  OverrideTest
The value of x at sub class is: 100
The value of y at sub class is: 200

D:\Java Programs>_
```

‘final’ Keyword In Java:

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

1. variable
2. method
3. class

1. **final variable:**

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

Ex: **final int** speedlimit=90;

2. final method: If we make any method as final, we cannot override it.

Ex:

```
    class Bike{
          final void run(){
                System.out.println("running");}
          }
    class Honda extends Bike{
          void run(){
                System.out.println("running safely with 100kmph");
           }
           public static void main(String args[]){
                Honda honda= new Honda();
                honda.run();
            }
     }
```

Output: Compile Time Error

3. final class: If we make any class as final, we cannot extend it.

Ex:

```
   final class Bike {
   }
   class Honda extends Bike{
     void run() {
           System.out.println("running safely with 100kmph");
           }
     public static void main(String args[]){
             Honda honda= new Honda();
             honda.run();
      }
   }
```

Output: Compile Time Error

Abstract Class in Java with example

A class that is declared using “**abstract**” keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods.

An abstract class cannot be **instantiated**, which means you are not allowed to create an **object**of it.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
   //This is abstract method
   abstract void myMethod();

   //This is concrete method with body
   void anotherMethod(){
      //Does something
   }
}
```

Abstract class Example

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
               System.out.println("Woof");
   }
   public static void main(String args[]){
               Dog  obj = new Dog();
               obj.sound();
   }
}
```

Output:

```
Woof
```

As we seen in the above example, there are cases when it is difficult or unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as **abstract**, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Interface

"An interface is a special case of abstract class, which contains all the final variables and abstract methods (methods without their implementation)."

An interface specifies what a class must do, but not how to do. The keyword ***interface*** is used to define interface. Once it is defined, any number of classes can implement an interface and also, one class can implement any number of interfaces.

Defining Interface:

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

interface interface_name {

variables declaration;

methods declaration;

}

Here, ***interface*** is the keyword and ***interface name*** is any valid Java variable (just like class name). All variables are declared as constants. Variables are declared as the following:

Static final type VariableName = Value;

Methods declaration will contain only a list of methods without any " ***body***" statements. The general format is

return-type methodName1 (parameter_list);

Ex:

```
interface Item {
        static final int code=1001;
        static final String name="Fan";
        void display();  //abstract method
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon. The class that implements this interface must define the code for the method.

Extending Interfaces:

Like classes, interfaces can also be extended i.e.,, an interface can be sub interfaced from other interfaces. The new sub interface will inherit all the members of the super interface in the manner similar to sub classes. This is achieved using the keyword extends as shown in below.

```
interface name2 extends name1 {
 body of name2
}
```

Ex:

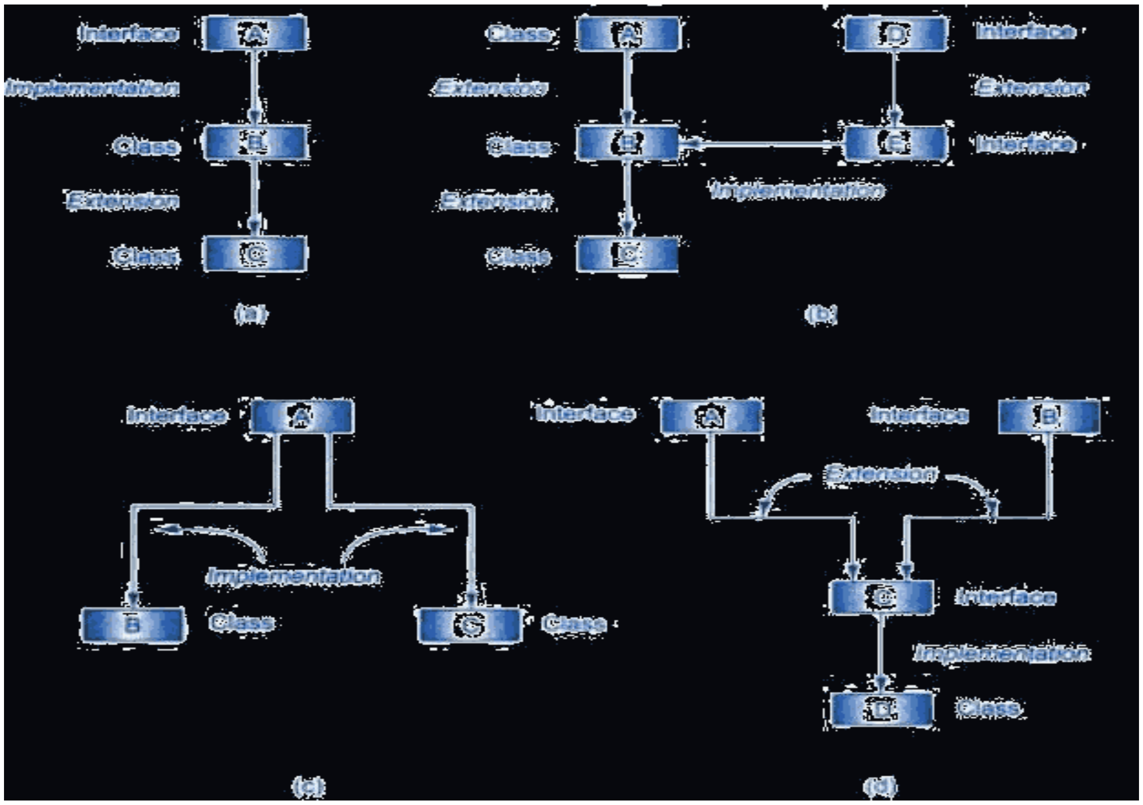
```
interface ItemConstants {
          int code=1001;
          String name="Fan";
}
interface Item extends ItemConstants {
          void display();
}
class Display implements Item {
          public void display( ) {
                    System.out.println(code + "  " + name);
          }
}
```

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 implements interfacename
{
          body of classname
}
```

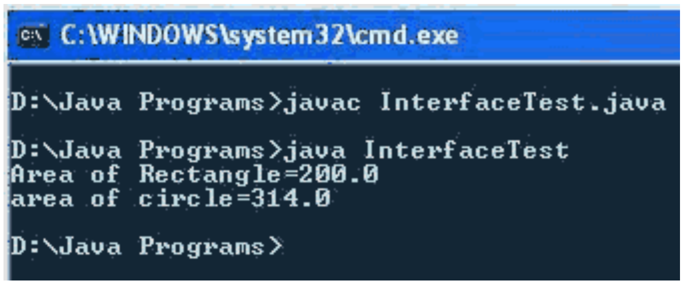
Here the class *classname* " ***implements***" the interface *interfacename*.When a class implements more than one interface, they are separated by a comma. The implementation of interfaces can take various forms as shown below:



***Ex: The following program illustrates the concept of interfaces in Java*.**

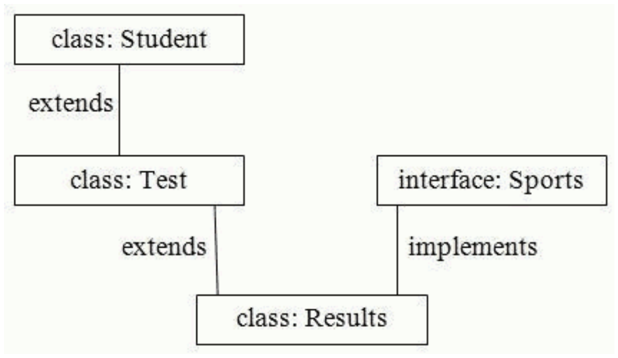
```
interface Area {          //Interface defined
        final static float pi=3.14f;
        float compute(float x,float y);
}
class Rectangle implements Area {
        public float compute(float x, float y) {
                return (x*y);
        }
}
class Circle implements Area {
        public float compute(float x,float y)  {
                return (pi*x*y);
        }
}
class InterfaceTest {
        public static void main(String args[]) {
                Rectangle rect=new Rectangle( );
                Circle cir=new Circle( );
                Area area;
                area=rect;
                System.out.println("Area of Rectangle=" + area.compute(10,20));
                area=cir;
                System.out.println("area of circle=" + area.compute(10,10));
        }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Programs>javac InterfaceTest.java

D:\Java Programs>java InterfaceTest
Area of Rectangle=200.0
area of circle=314.0

D:\Java Programs>
```

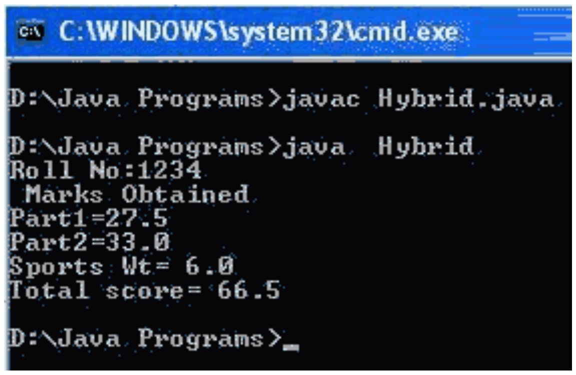
Multiple Inheritance: Multiple Inheritance consists multiple base classes and a single derived class. To implement this concept not only using classes, but also required interface in Java.



Ex:

```
        class Student {
                int rno;
        void getNumber(int n) {
                rno=n;
        }
        void putNumber( ) {
                System.out.println("Roll No:" + rno);
        }
}
class Test extends Student {
        float part1,part2;
        void getMarks(float m1,float m2) {
                part1=m1;
                part2=m2;
        }
        void putMarks( ) {
                System.out.println("Marks Obtained");
                System.out.println("Part1=" + part1);
                System.out.println("Part2=" + part2);
        }
}

interface Sports {
        float sportWt=6.0f;
        void putWt();
}
class Results extends Test implements Sports {
        float total;
        public void putWt( )   {
                System.out.println("Sports Wt= " + sportWt);
        }
        void display( ) {
                total=part1+part2+sportWt;
                putNumber();
                putMarks();
                putWt();
                System.out.println("Total score= " + total);
        }
}
class Hybrid {
        public static void main(String args[]) {
                Results std1=new Results();
                std1.getNumber(1234);
                std1.getMarks(27.5f, 33.0f);
                std1.display();
        }
}
```

Output:



```
D:\Java Programs>javac Hybrid.java

D:\Java Programs>java  Hybrid
Roll No:1234
 Marks Obtained
Part1=27.5
Part2=33.0
Sports Wt= 6.0
Total score= 66.5

D:\Java Programs>_
```

Interface Vs Abstract Classes in Java

There are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.

3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){
System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the metho
ds
class M extends B{
public void a(){
```

```
System.out.println("I am a");
}
public void b(){
System.out.println("I am b");
}
public void d(){
System.out.println("I am d");
}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Test it Now

Output:

```
I am a
I am b
I am c
I am d
```

Exceptions & Assertions

"An Exception is an abnormal error condition that arises during our program execution" .

When an Exception occurs in a program, the Java interpreter creates an exception object and throws it out as Java exceptions, which are implemented as objects of exception class. This class is defined in j***ava.lang*** package. An Exception object contains data members that will store the exact information about the runtime error (Exception) that has occurred.

Types of Errors:

Errors may broadly classify into two categories.

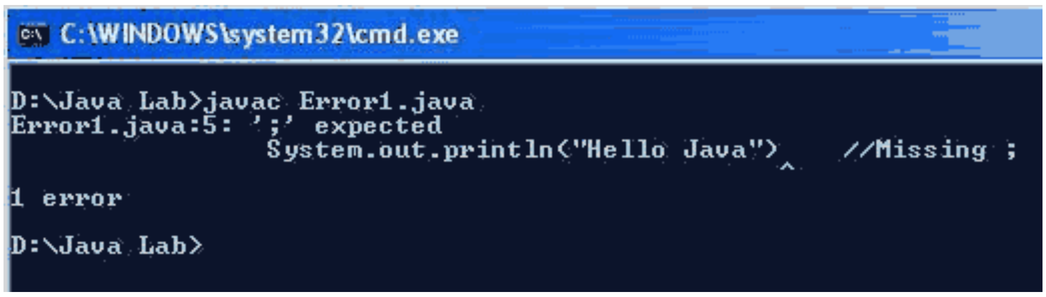
1. Compile-time errors
2. Run-time errors.

Compile-Time Errors:

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as ***compile-time errors***. Whenever the compiler displays an error, it will not create the ***.class*** file. The following program is the illustration of Compile-time errors:

```
class Error1 {
        public static void main(String args[]) { System.out.println("Hello
                Java") //Missing ;
        }
}
```

When we compile the above program, the Java compiler displays the following error:



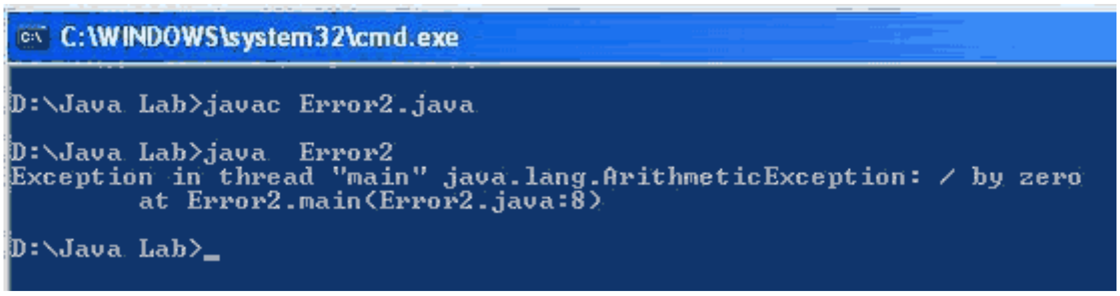
```
C:\WINDOWS\system32\cmd.exe

D:\Java Lab>javac Error1.java
Error1.java:5: ';' expected
                System.out.println("Hello Java")    //Missing ;
                                                ^
1 error

D:\Java Lab>
```

Runtime Errors:

Sometimes, a program may compile successfully creating the ***.class*** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. The following program is the illustration of Run-time errors .

```
class Error2 {
        public static void main(String[] args) {
                int a=10;
                int b=5;
                int c=5;
                int x=a/(b-c);    //Division by zero
                System.out.println("x=" + x);
                int y=a/(b+c);
                System.out.println("y=" + y);
        }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Lab>javac Error2.java

D:\Java Lab>java  Error2
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Error2.main(Error2.java:8)

D:\Java Lab>_
```

Exception Handling:

An ***exception*** is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (informs us that an error occurs).

If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the above output and will terminate the program.

If we want the program to continue with the execution of the remaining code, then we should try to catch the object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as ***Exception Handing***. The following table shows the some common errors that are occurred in the Java programs.

S.No	Exception Type	Cause of Exception
1	Arithmetic Exception	Caused by the math errors such as division by zero.
2	ArrayIndexOutOfBoundsException	Caused by bad array indexes.
3	FileNotFoundException	Caused by an attempt to access a non existing file.
4	NumberFormatException	Caused when a conversion between strings and numbers fails.
5	NullPointerException	Caused by referencing a null object.

Exceptions in Java can be categorized into two types.

1. Checked Exceptions:

These exceptions are explicitly handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from the j***ava.lang.Exception*** class.

2. Unchecked Exceptions:

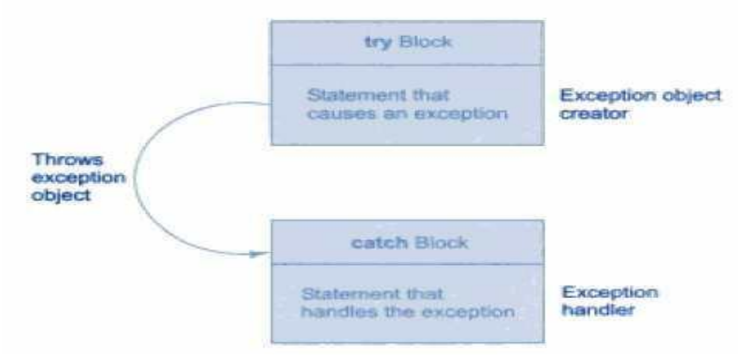
These exceptions are not essentially handled in the program code, instead the JVM handles such exceptions. Unchecked exceptions are extended from the class j***ava.lang.RuntimeException***.

Syntax of Exception Handling Code:

The basic concepts of exception handling are throwing an exception and catching it.

The following diagram illustrates this concept:

Java uses the keywords ***try*** and ***catch*** to handle the exceptions in the programs.



try block:

The statements that produce exception are identified in the program and the statements are placed in *try block*. The general format is

try {

//Statements that causes Exception

}

catch block:

The catch block is used to process the exception raised. The catch block is placed immediately after the try block. The general format is

catch(ExceptionType ex_ob) {

//Statements that handle Exception

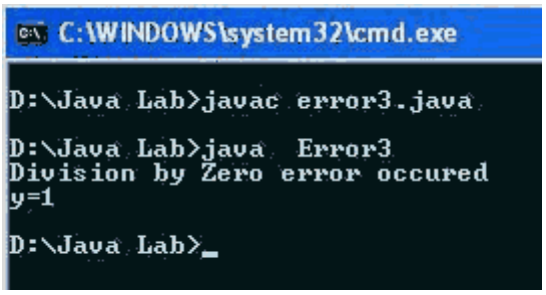
}

The following program illustrates the use of using the ***try*** and ***catch*** blocks in the Java programs:

```
class Error3 {   a//
    public static void main(String args[]) {
            int a=10;
            int b=5;
            int c=5;
            int x,y;
```

```
                try {
                        x=a/(b-c);    //Division by zero
                        System.out.println("x=" + x);
                }
                catch(ArithmeticException e) {
                        System.out.println("Division by Zero error occured");
                }
                y=a/(b+c);
                System.out.println("y=" + y);
        }
    }
```

Output:



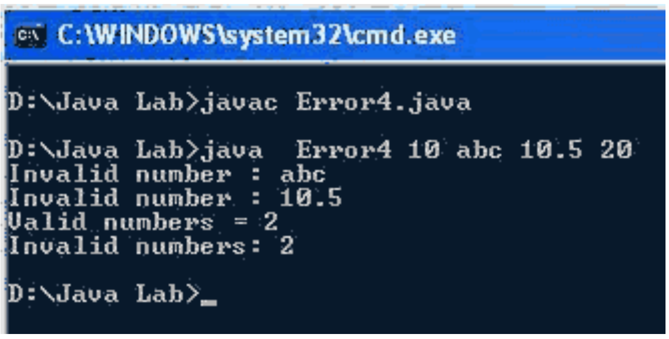
```
C:\WINDOWS\system32\cmd.exe

D:\Java Lab>javac error3.java

D:\Java Lab>java  Error3
Division by Zero error occured
y=1

D:\Java Lab>_
```

The above program did not stop at the point of exception condition. It catches the error condition and prints the error message.

The following is the program that catches the invalid command line arguments:

```
class Error4 {
        public static void main(String args[]) {
                int number,invalid=0, valid=0;
                for(int i=0;i<args.length;i++) {
                        try {
                                number=Integer.parseInt(args[i]);
                        }
                        catch(NumberFormatException e) {
                                invalid=invalid+1;
                                System.out.println("Invalid number : " + args[i]);
                                continue;
                        }
                        valid=valid+1;
                }
                System.out.println("Valid numbers = " + valid);
                        System.out.println("Invalid numbers: " + invalid);
        }
}
```

Output:


C:\WINDOWS\system32\cmd.exe

```
D:\Java Lab>javac Error4.java

D:\Java Lab>java  Error4 10 abc 10.5 20
Invalid number : abc
Invalid number : 10.5
Valid numbers = 2
Invalid numbers: 2

D:\Java Lab>_
```

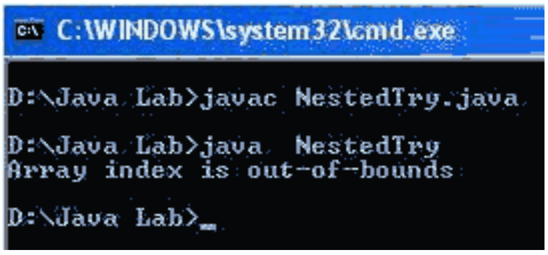
Nested Try:

A try block is placed inside the block of another try block is termed as ***Nested try block*** statements. If any error statement is in outer try block, it goes to the corresponding outer catch block. If any error statement is in inner try block, first go to the inner catch block. If it is not the corresponding exception, next goes to the outer catch, if it is also not the corresponding exception then it will be terminated.

Ex:

```
class NestedTry {
        public static void main(String args[]) {
                try {
                        int a=2,b=4,c=2,x=7,z;
                        int p[]={2};
                        p[3]=33;
                        try {
                                z=x/((b*b)-(4*a*c));
                                System.out.println("The value of x is= " + z);
                        }
                        catch (ArithmeticException e) { System.out.println("Division by zero in
                                arithmetic expression");
                        }
                }
                catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array
                        index is out-of-bounds");
                }
        }
}
```

Output:


C:\WINDOWS\system32\cmd.exe

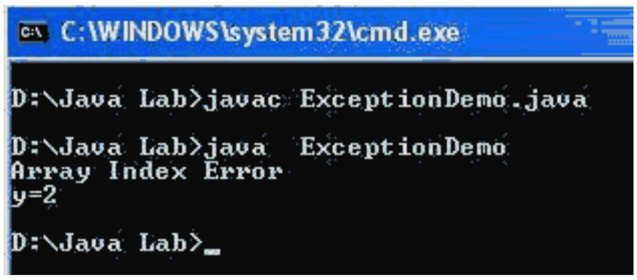
```
D:\Java Lab>javac NestedTry.java

D:\Java Lab>java  NestedTry
Array index is out-of-bounds

D:\Java Lab>_
```

Multiple Catch Statements: Multiple catch statements handle the situation where more than one exception could be raised by a single piece of code. In such situations specify two or more catch blocks, each specify different type of exception.

Ex:

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

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Lab>javac ExceptionDemo.java

D:\Java Lab>java  ExceptionDemo
Array Index Error
y=2

D:\Java Lab>_
```

Finally Statement:

finally creates a block of code that will be executed after a ***try/catch*** block has completed. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. The general format is

```
finally {
        // statements that executed before try/catch
 }
```

Ex:

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

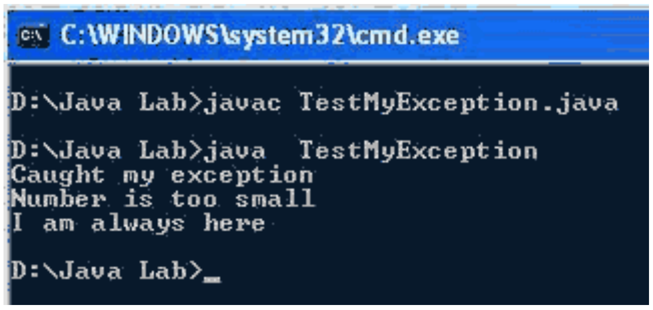
Throwing Our Own Exceptions (User Defined Exceptions):

It is possible to create our own exception types to handle situations specific to our application. Such exceptions are called User-defined Exceptions. User defined exceptions are created by extending **Exception** class. The **throw** and **throws** keywords are used while implementing *user-defined* exceptions.

Ex:

```
class MyException extends Exception {
        MyException(String msg) {
                super(msg);
        }
}
class TestMyException {
        public static void main(String args[]) {
                int x=5,y=1000;
                try {
                        float z=(float)x/(float)y;
                        if(z<0.01) {
                                throw new MyException("Number is too small");
                        }
                }
                catch(MyException e) {
                        System.out.println("Caught my exception");
                        System.out.println(e.getMessage());
                }
                finally {
                        System.out.println("I am always here");
                }
                }
        }
```

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Lab>javac TestMyException.java

D:\Java Lab>java  TestMyException
Caught my exception
Number is too small
I am always here

D:\Java Lab>_
```

Exception Encapsulation, Enrichment and Assertions:

Exception encapsulation and enrichment

The process of wrapping the caught exception in a different exception is called "Exception Encapsulation". The Throwable super class has added one parameter in its constructor for the wrapped exception and a "getCause()" method to return the wrapped exception. Wrapping is also used to hide the details of implementation.

Syntax:

```
         try
        {
                throw new ArithmeticException();
        }
        catch(AritmeticExceptions ae)
        {
        //wrapping exception
                throw new ExcepDemo("Testing User Exception",ae);
        }
```

Disadvantages of wrapping:

1. It leads to the long Stack traces.
2.It is a difficult task to figure out where the exception is Solution:

The possible solution is Exception enrichment. Here we don't wrap exception but we add some information to the already thrown exception and rethrow it.
Example program:

```
class ExcepDempo extends Exception
{
        String message;
        ExcepDemo(String msg)
        {
                message=msg;
        }
        public void addInformation(String msg)
        {
                message=message+msg;
        }
}
class ExcepEnrich
```

```
{
        static void testException()
        {
                try{
                        throw new ExcepDemo("Testing user Exception:");
                }
```

```
            catch(ExcepDemo e)
            {
                e.addInformation("Example Exception");
            }
        }
        public static void main(String args[])
        {
            try
        {
            testException();
        }
        catch(Exception e)
        {


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

Assertions

Assertions are added after java 1.4 to always create reliable programs that are Correct and robust programs. The assertions are Boolean expressions. Conditions such as positive number or negative number are examples.

Syntax:

assert expression1;

or

assert expression1:expression2;

Where **assert** is the keyword. Expression 1 is the Boolean expression, expression2 is the string that describes the Exceptions.
Note: assertions must be explicitly enabled. The –ea option is used to enable the exception and –da is used to disable the exception.

AI.java

```
import java.io.*;
class AI
{
 void check(int i)
 {
        assert i>0:" I must be positive:";
        System.out.println("Your I value is fine");
 }
 public static void main(String args[]) throws IOException
 {
        AI a=new AI();
        a.check(Integer.parseInt(args[0]));
 }
}}
```

Packages

Package: ***"A Package is a collection of classes, interfaces and sub-packages."***

A Sub package in turns divides into classes, interfaces and sub-sub-packages, etc., learning about JAVA is nothing but learning about various packages. By default one predefined package is imported for each and every java program and whose name is java.lang.*.

Advantages:

1. The classes contained in the packages of other programs/applications can be reused.
2. Packages classes can be unique compared with classes in other packages.
3. Classes in packages can be hidden if we don't want other packages to access them.
4. Packages also provide a way for separating "design" from coding.

In Java the packages are classified into two categories. They are

1. Java API packages (Predefined Packages)
2. User defined Packages

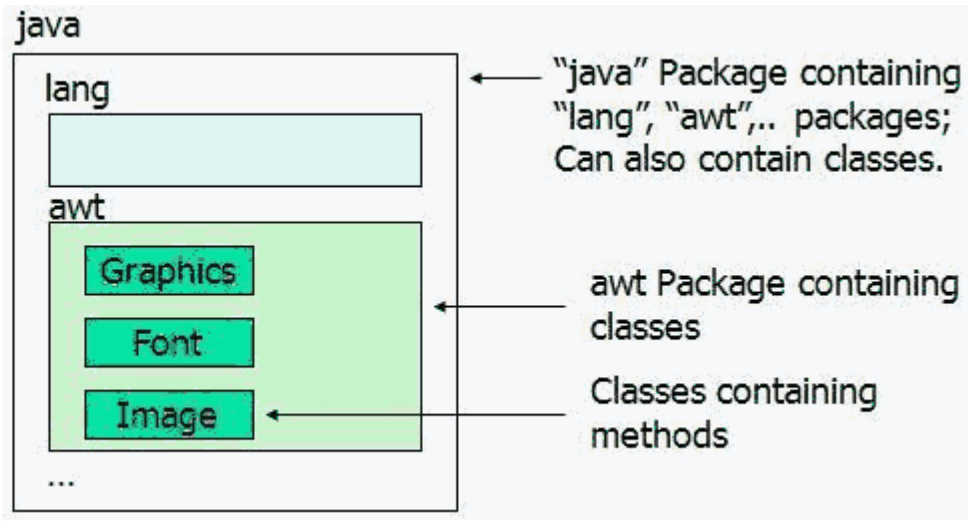
Java API packages

Java API (**A**pplication **P**rogramming **I**nterface) provides a large number of classes grouped into different packages according to their functionalities. The generally used packages are:

Package Name	Package Description
java.lang	Contains classes for primitive types, strings, math functions, threads, and exception.
java.util	Contains classes such as vectors, hash tables, date, scanner etc.
java.io	Stream classes for I/O
java.awt	Classes for implementing GUI – windows, buttons, menus etc. (AWT: ***A***bstract ***W***indow ***T***oolkit)
java.net	Classes for networking
java.applet	Classes for creating and implementing applets

Using System Packages:

The packages are organized in a hierarchical structure. For example, a package named "java" contains the package "awt", which in turn contains various classes required for implementing GUI (graphical user interface).


java
lang
awt
Graphics
Font
Image
...
"java" Package containing "lang", "awt",.. packages; Can also contain classes.
awt Package containing classes
Classes containing methods

Java.lang Package:

The package `java.lang` contains classes and interfaces that are essential to the Java language. These include:

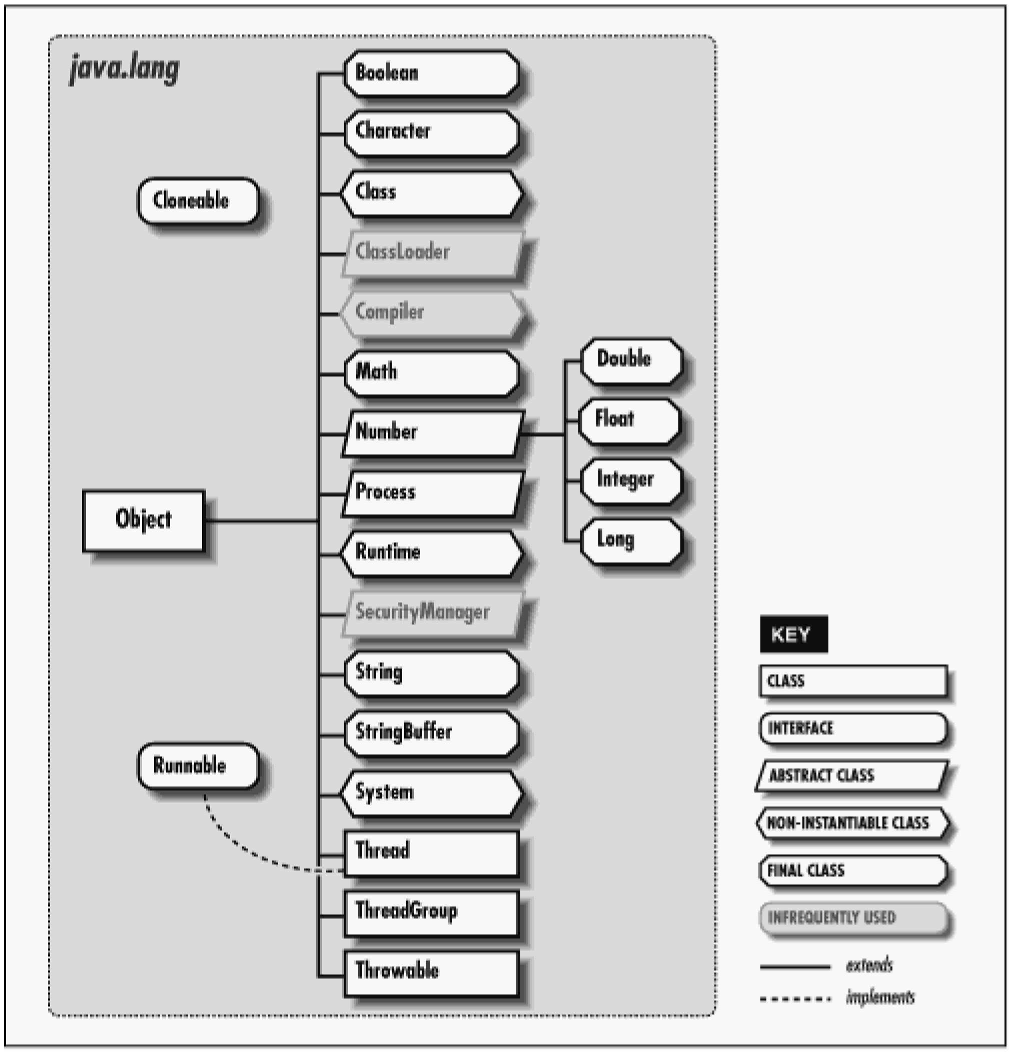
- Object, the ultimate superclass of all classes in Java
- Thread, the class that controls each thread in a multithreaded program

- Throwable, the superclass of all error and exception classes in Java
- Classes that encapsulate the primitive data types in Java
- Classes for accessing system resources and other low-level entities
- Math, a class that provides standard mathematical methods
- String, the class that is used to represent strings

Because the classes in the `java.lang` package are so essential, the `java.lang` package is implicitly imported by every Java source file. In other words, you can refer to all of the classes and interfaces in `java.lang` using their simple names.

The following figure shows the class hierarchy for the `java.lang` package.

Figure 10.1: The java.lang package



Creating Packages:

Java supports a keyword called " ***package***" for creating user-defined packages. The package statement must be the first statement in a Java source file (except comments and white spaces) followed by one or more classes.

```
                 package myPackage;
                public class ClassA {
                        // class body
        }
        class ClassB {
          //    class body
        }
```

Where ***package name*** is " ***myPackage***" and ***classes*** are considered as part of this package. The code is saved in a file called " ***ClassA.java***" and located in a directory called " ***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.** 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

To create our own package involves the following steps:

1. Declare the package at the beginning of a file using the form:

   package *packagename***;**

2. Define the class that is to be put in the package and declare it ***public***.
3. Create a subdirectory under the directory where main source files are stored.
4. Store the listing as the ***classname.java*** file in the subdirectory created.
5. Compile the file. This creates ***.class*** file in the subdirectory.

Accessing a Package:

Java system package can be accessed either using a fully qualified class name or using a shortcut approach through the import statement. The general form of import statement for searching a ***class*** is as follows:

import package1 [.package2] [.package3].classname;

Where ***package1*** is the name of the top level package, ***package2*** is name of the package that inside the ***package1*** and so on.

Using a Package:

The following steps explain how to create user-defined packages and use them in the programs.

1. Create a directory named as ***package1***.
2. Type the following code in a file and save it as ***ClassA.java*** in the ***package1*** subdirectory.

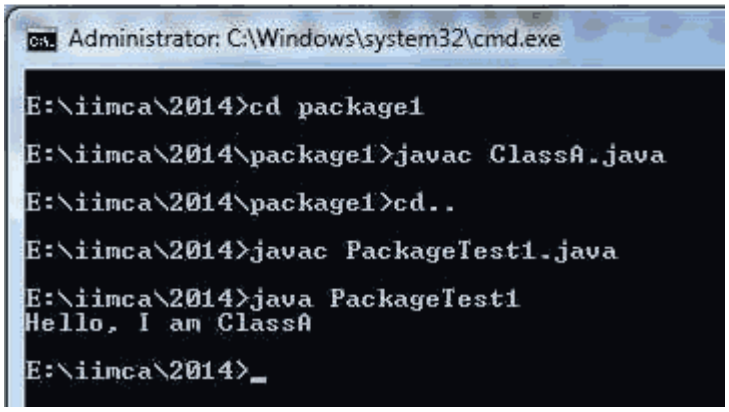
```
package package1;
public class ClassA {
        public void displayA( ) {
           System.out.println("Hello, I am ClassA");
        }
}
```

We should declare ***ClassA*** with access specifier ***public***.

3. Compile it.
4. Type the following code in a file and save it as ***PackageTest1.java*** in the outside of ***package1*** subdirectory.

```
import package1.ClassA;
class PackageTest1 {
        public static void main(String args[]) {
                ClassA objA=new ClassA( );
                objA.displayA( );
        }
}
```

5. Compile and execute it. We will get the output as the following:



```
E:\iimca\2014>cd package1

E:\iimca\2014\package1>javac ClassA.java

E:\iimca\2014\package1>cd..

E:\iimca\2014>javac PackageTest1.java

E:\iimca\2014>java PackageTest1
Hello, I am ClassA

E:\iimca\2014>_
```

6. To create one more package, create a subdirectory ***package2***.

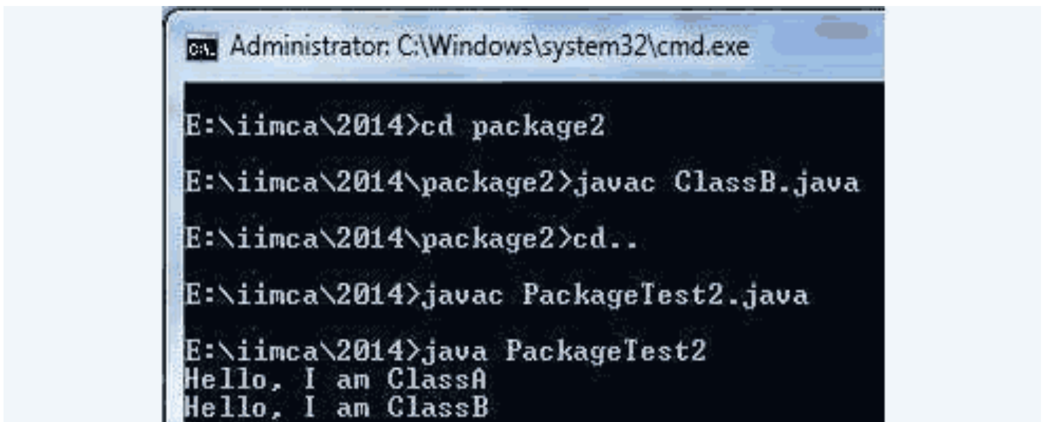
7. Type the following code in a file and save it as ***ClassB.java*** in the ***package2*** subdirectory.

```
package package2;
public class ClassB {
        protected int m=10;
        public void displayB( )  {
           System.out.println("Hello, I am ClassB");
            System.out.println("m = " + m);
        }
}
```

8. Type the following code in a file and save it as ***PackageTest2.java*** in the outside of ***package2*** subdirectory.

```
import package1.ClassA;
import package2.*;
class PackageTest2 {
        public static void main(String args[]) {
                ClassA objA=new ClassA( );
                ClassB objB=new ClassB( );
                objA.displayA( );
                objB.displayB( );
        }
}
```

8. Compile and execute it. We will get the output as the following.



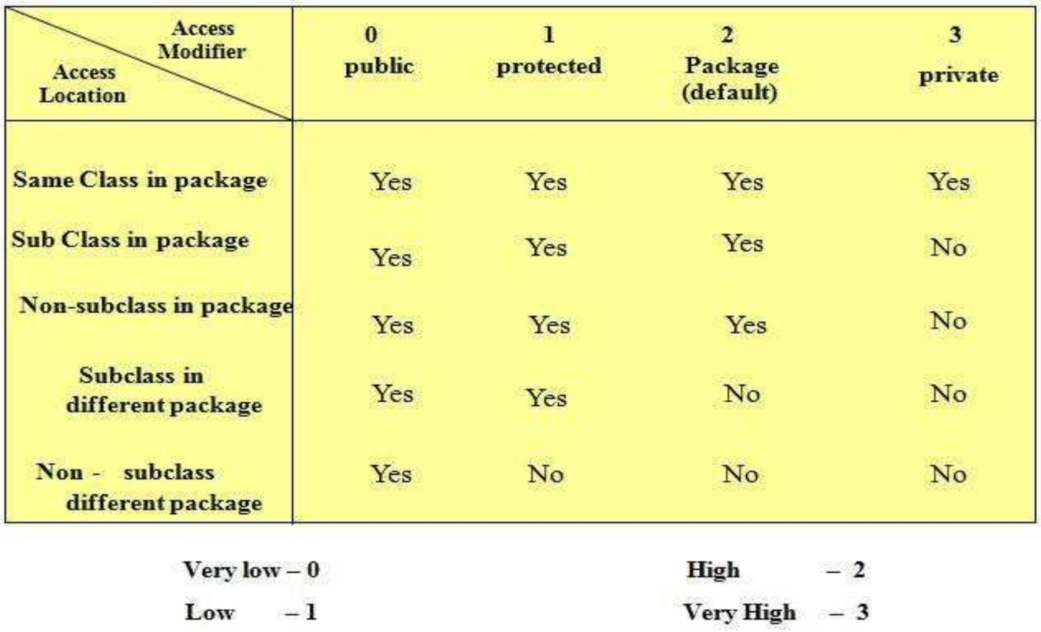
Administrator: C:\Windows\system32\cmd.exe

```
E:\iimca\2014>cd package2

E:\iimca\2014\package2>javac ClassB.java

E:\iimca\2014\package2>cd..

E:\iimca\2014>javac PackageTest2.java

E:\iimca\2014>java PackageTest2
Hello, I am ClassA
Hello, I am ClassB
```

Access Protection (Visibility Modifiers)

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.



Access Modifier / Access Location	0 public	1 protected	2 Package (default)	3 private
Same Class in package	Yes	Yes	Yes	Yes
Sub Class in package	Yes	Yes	Yes	No
Non-subclass in package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non - subclass different package	Yes	No	No	No

Very low – 0	High – 2
Low – 1	Very High – 3

MULTI THREADING

JAVA.LANG.THREAD: The **java.lang.Thread** class is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Following are the important points about Thread −

- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority
- There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread and,
- the other way to create a thread to declare a class that implements the Runnable interface

Class Declaration

Following is the declaration for **java.lang.Thread** class −

```
public class Thread
   extends Object
      implements Runnable
```

Fields

Following are the fields for **java.lang.Thread** class −

- **static int MAX_PRIORITY** − This is the maximum priority that a thread can have.
- **static int NORM_PRIORITY** − This is the default priority that is assigned to a thread.

Class constructors

Sr.No.	Constructor & Description
1	**Thread()** This allocates a new Thread object.
2	**Thread(Runnable target)** This allocates a new Thread object.
3	**Thread(Runnable target, String name)** This allocates a new Thread object.
4	**Thread(String name)** This constructs allocates a new Thread object.
5	**Thread(ThreadGroup group, Runnable target)** This allocates a new Thread object.

6	**Thread(ThreadGroup group, Runnable target, String name)** This allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
7	**Thread(ThreadGroup group, Runnable target, String name, long stackSize)** This allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified stack size.
8	**Thread(ThreadGroup group, String name)** This allocates a new Thread object.

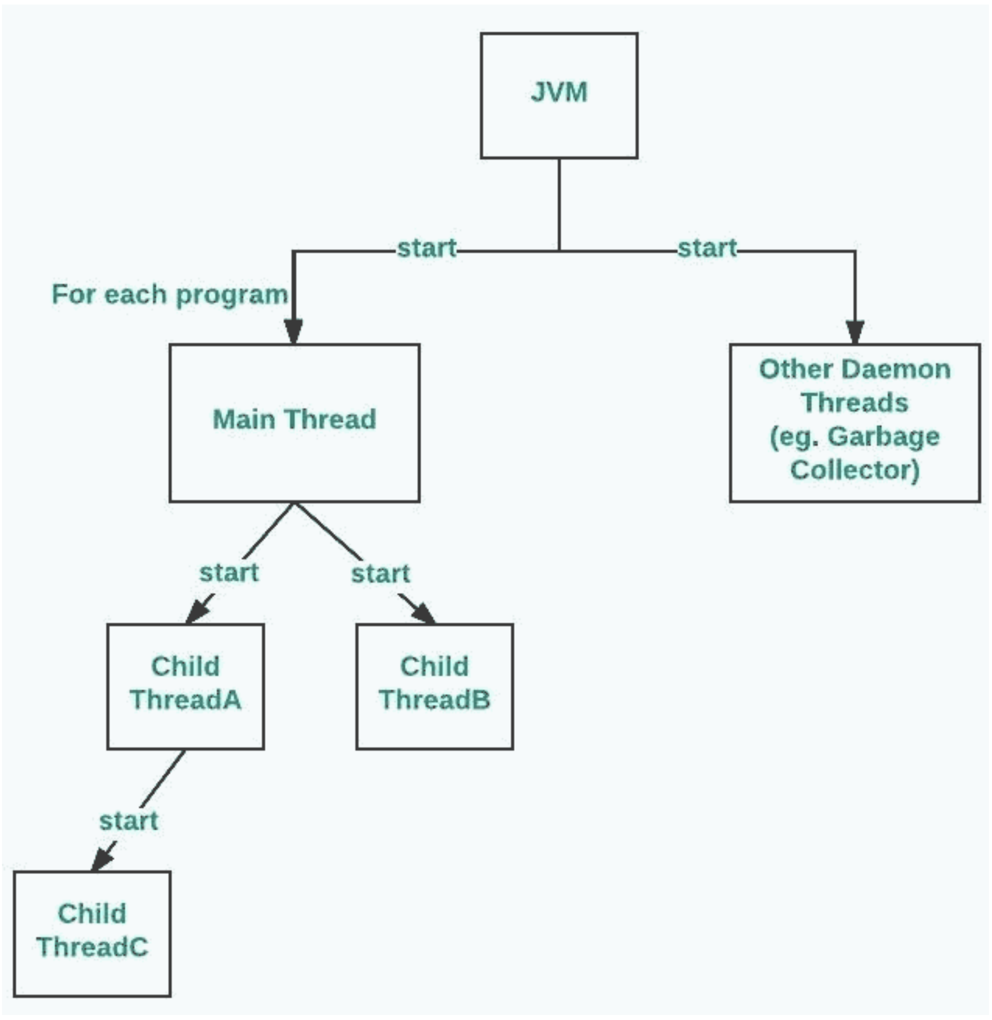
THE MAIN THREAD:

When a Java program starts up, one thread begins running immediately. This is usually called the ***main*** thread of our program, because it is the one that is executed when our program begins.

Properties :

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions

Flow diagram :



The main thread is created automatically when our program is started. To control it, we must obtain a reference to it. This can be done by calling the method *currentThread()*which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

// Java program to control the Main Thread

```
public class Test extends Thread
{
   public static void main(String[] args)
   {
       // getting reference to Main thread
       Thread t = Thread.currentThread();
        // getting name of Main thread
       System.out.println("Current thread: " + t.getName());
      // changing the name of Main thread
       t.setName("Geeks");
       System.out.println("After name change: " + t.getName());
```

```
        // getting priority of Main thread
         System.out.println("Main thread priority: "+ t.getPriority());
        // setting priority of Main thread to MAX(10)
         t.setPriority(MAX_PRIORITY);
        System.out.println("Main thread new priority: "+ t.getPriority());
        for (int i = 0; i < 5; i++)
         {
             System.out.println("Main thread");
         }
         // Main thread creating a child thread
         ChildThread ct = new ChildThread();
         // getting priority of child thread
         // which will be inherited from Main thread
         // as it is created by Main thread
         System.out.println("Child thread priority: "+ ct.getPriority());
        // setting priority of Main thread to MIN(1)
         ct.setPriority(MIN_PRIORITY);
        System.out.println("Child thread new priority: "+ ct.getPriority());
        // starting child thread
         ct.start();
    }
}
  // Child Thread class
class ChildThread extends Thread
{
    @Override
    public void run()
    {
        for (int i = 0; i < 5; i++)
        {
            System.out.println("Child thread");
        }
    }
}
```

Output:

```
Current thread: main
```

```
After name change: Geeks
Main thread priority: 5
Main thread new priority: 10
Main thread
Main thread
Main thread
Main thread
Main thread
Child thread priority: 10
Child thread new priority: 1
Child thread
Child thread
Child thread
Child thread
Child thread
```

CREATION OF NEW THREADS :

Threads are implemented in the form of objects that contain a method called **run( ).** The **run( )** method is the heart and soul of any thread. A typical **run( )** would appear as follows.

```
public void run( ) {
    …………….
    …………….     (statements for implementing thread)
    …………….
}
```

The run ( ) method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start ( ).** A new thread can be created in two ways:

1. By **extending Thread** class: Define a class that extends **Thread** class and override its run( ) method with the code required by the thread.
2. By **implementing Runnable** interface: Define a class that implements Runnable interface. The Runnable interface has only one method, run (), that is to be defined in the method with the code to be executed by the thread.

1. Extending Thread Class:

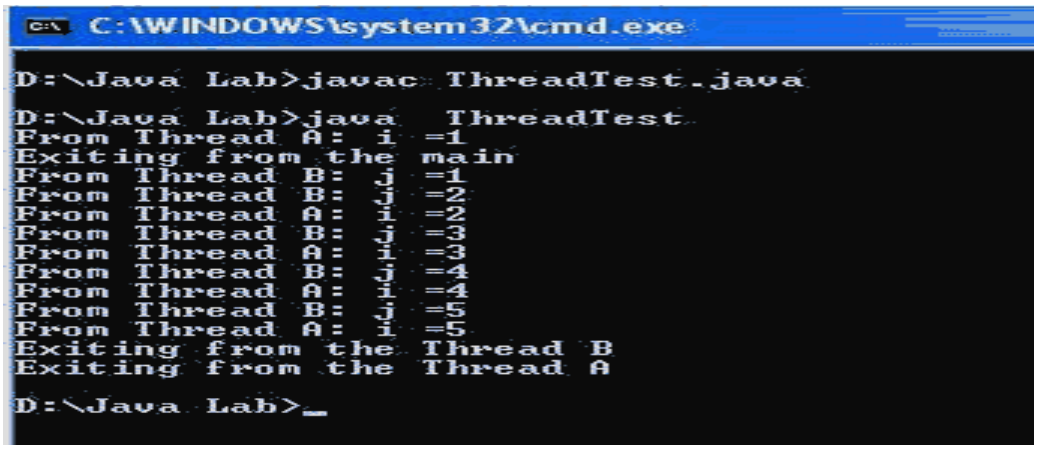
We can make our class ***runnable*** as thread by extending the class ***java.lang.Thread***. This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the ***Thread*** class.
2. Implement the ***run ( )*** method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the ***start ( )*** method to initiate the thread execution.

The following program illustrates the concept of extending the thread class.

```
class  A extends Thread {
    public void run( ) {
            for(int i=1;i<=5;i++)  {
                    System.out.println("From Thread A: i =" + i);
            }
            System.out.println("Exiting from the Thread A");
    }
}
class  B extends Thread {
    public void run( ) {
            for(int j=1;j<=5;j++) {
                    System.out.println("From Thread B: j =" + j);
            }
            System.out.println("Exiting from the Thread B");
    }
}
class ThreadTest {
    public static void main(String args[]) {
            A a=new A();
            B b=new B();
            a.start();
            b.start();
            System.out.println("Exiting from the main");
    }
}
```

Output:



```
D:\Java Lab>javac ThreadTest.java

D:\Java Lab>java  ThreadTest
From Thread A: i =1
Exiting from the main
From Thread B: j =1
From Thread B: j =2
From Thread A: i =2
From Thread B: j =3
From Thread A: i =3
From Thread B: j =4
From Thread A: i =4
From Thread B: j =5
From Thread A: i =5
Exiting from the Thread B
Exiting from the Thread A

D:\Java Lab>_
```

2.Implementing Runnable Interface:

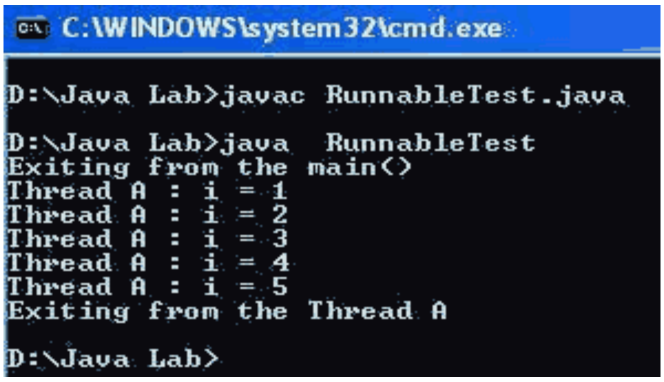
The Runnable interface declares the run ( ) method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

1. Declare the class as implementing the Runnable interface.
2. Implement the ***run( )*** method.
3. call the thread's ***start( )*** method to run the thread.

The following program illustrates the concept of implementing the Runnable Interface in Java.

```
    class A implements Runnable {
            public void run( ) {
                for(int i=1;i<=5;i++)  {
                        System.out.println("Thread A : i = " + i);
                }
                    System.out.println("Exiting from the Thread A");
            }
    }
    class RunnableTest {
            public static void main(String args[]) {
                A obj=new A( );
                Thread t=new Thread(obj);
                t.start( );
                    System.out.println("Exiting from the main( )");
            }
    }
```

Output:



```
C:\WINDOWS\system32\cmd.exe

D:\Java Lab>javac RunnableTest.java

D:\Java Lab>java  RunnableTest
Exiting from the main()
Thread A : i = 1
Thread A : i = 2
Thread A : i = 3
Thread A : i = 4
Thread A : i = 5
Exiting from the Thread A

D:\Java Lab>
```

Lifecycle of a Thread:

During the lifetime of a thread, there are different states it can enter. They include:

1. NewBorn State

2. Runnable State
3. Running State
4. Blocked State
5. Dead State

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in below figure.

1. NewBorn State:

When we create a thread object, the thread is born and is said to be in ***newborn*** state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

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

2. Runnable State:

The ***runnable*** state means that the thread is ready for execution and is waiting for the availability of the processor. If we want a thread to hand over control to another thread to equal priority before its turn comes, we can do it by using the **yield( ).**

3. Running State:

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

4. Blocked State:

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered “not Runnable” but not dead and therefore fully qualified to run again.

5. Dead State:

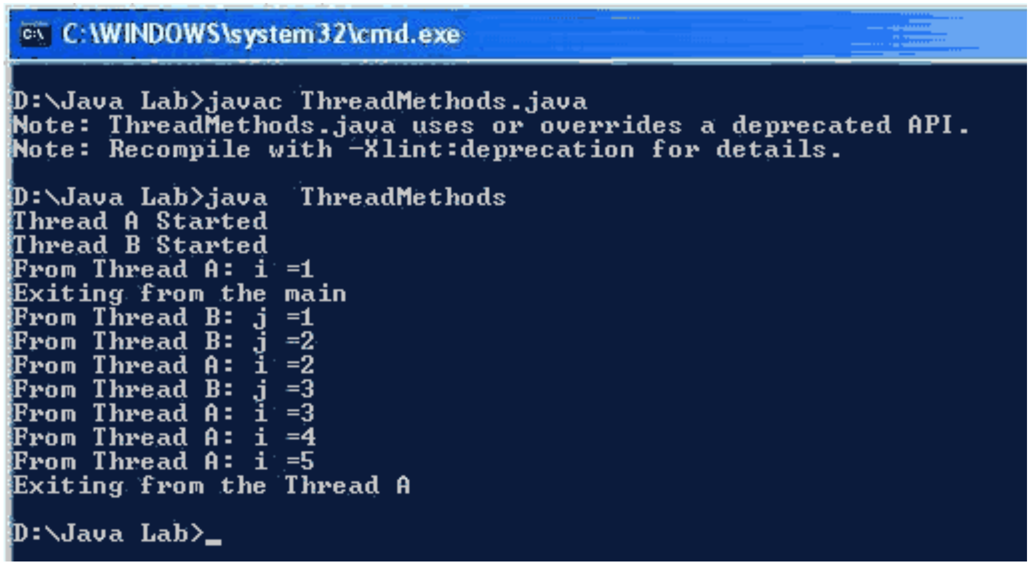
Every thread has a lifecycle. A running thread ends its life when it has completed executing its run( ) method. It is natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. It is done by stop( ) method.

The following program illustrates the concept of using the thread methods.

```
class  A extends Tread {
   public void run( ) {
               for(int i=1;i<=5;i++) {
                    if(i==1)
                            yield();
                    System.out.println("From Thread A: i =" + i);
            }
```

```
                System.out.println("Exiting from the Thread A");
     }
  }
  class  B extends Thread {
     public void run( ) {
                for(int j=1;j<=5;j++) {
                          System.out.println("From Thread B: j =" + j);
                          if(j==3)
                                    stop();
                }
                System.out.println("Exiting from the Thread B");
     }}
  class ThreadMethods {
     public static void main(String args[]) {
                A a=new A();
                B b=new B();
                System.out.println("Thread A Started");
                a.start();
                System.out.println("Thread B Started");
                b.start();
                System.out.println("Exiting from the main");
     }
  }
```

Output:



C:\WINDOWS\system32\cmd.exe

```
D:\Java Lab>javac ThreadMethods.java
Note: ThreadMethods.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

D:\Java Lab>java  ThreadMethods
Thread A Started
Thread B Started
From Thread A: i =1
Exiting from the main
From Thread B: j =1
From Thread B: j =2
From Thread A: i =2
From Thread B: j =3
From Thread A: i =3
From Thread A: i =4
From Thread A: i =5
Exiting from the Thread A

D:\Java Lab>_
```

THREAD PRIORITY:

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The Threads of the same priority are given equal treatment by the java scheduler and, therefore, they share the processor on a First-Come, First-Serve basis.

Java permits us to set the priority of a thread using the ***setPriority( )*** method as follows:

ThreadName.setPriority(Number);

The Number is an integer value to which the threads priority is set. The Thread class defines several priority constants.

MIN_PRIORITY=1

NORM_PRIORITY=5

MAX_PRIORITY=10

Whenever multiple Threads are ready for execution, the Java system chooses the highest priority thread and executes it. For a thread of lower priority to gain control, one of the following things should happen.

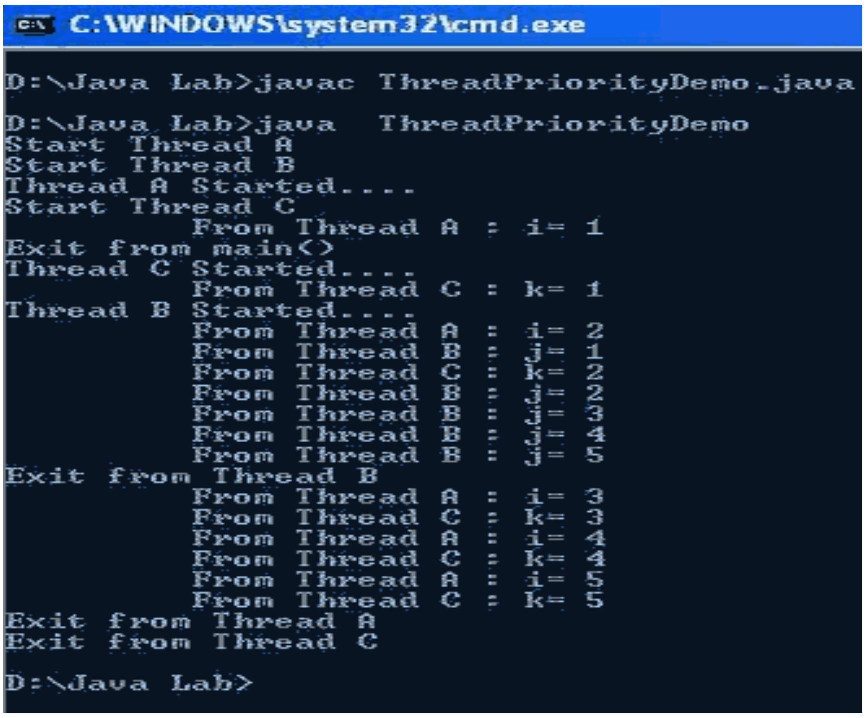
1. It stops running at the end of run( ).
2. It is made to sleep using sleep( ).
3. It is told to wait using wait( ).

The following program illustrates the concept of thread priorities in Java.

```
class A extends Thread {
        public void run( )      {
                System.out.println("Thread A Started....");
                for(int i=1;i<=5;i++)  {
                                System.out.println("\t From Thread A : i= " + i);
                }
                System.out.println("Exit from Thread A");
        }
}
class B extends Thread {
        public void run( ) {
                System.out.println("Thread B Started....");
                for(int j=1;j<=5;j++)  {
                                System.out.println("\t From Thread B : j= " + j);
                }
                System.out.println("Exit from Thread B");
        }
}
```

```
class C extends Thread {
        public void run( ) {
                System.out.println("Thread C Started....");
                for(int k=1;k<=5;k++) {
                                System.out.println("\t From Thread C : k= " + k);
                }
                System.out.println("Exit from Thread C");
        }
}
class ThreadPriorityDemo {
        public static void main(String args[]) {
                A threadA=new A();
                B threadB=new B();
                C threadC=new C();
                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.setPriority(threadA.getPriority()+1);
                threadA.setPriority(Thread.MIN_PRIORITY);
                System.out.println("Start Thread A");
                threadA.start();
                System.out.println("Start Thread B");
                threadB.start();
                System.out.println("Start Thread C");
                threadC.start();
                System.out.println("Exit from main()");
        }
}
```

Output:



C:\WINDOWS\system32\cmd.exe

```
D:\Java Lab>javac ThreadPriorityDemo.java

D:\Java Lab>java  ThreadPriorityDemo
Start Thread A
Start Thread B
Thread A Started....
Start Thread C
         From Thread A : i= 1
Exit from main()
Thread C Started....
         From Thread C : k= 1
Thread B Started....
         From Thread A : i= 2
         From Thread B : j= 1
         From Thread C : k= 2
         From Thread B : j= 2
         From Thread B : j= 3
         From Thread B : j= 4
         From Thread B : j= 5
Exit from Thread B
         From Thread A : i= 3
         From Thread C : k= 3
         From Thread A : i= 4
         From Thread C : k= 4
         From Thread A : i= 5
         From Thread C : k= 5
Exit from Thread A
Exit from Thread C

D:\Java Lab>
```

MULTI THREADING – USING IS ALIVE() AND JOIN()

In java, **isAlive()**and **join()** are two different methods that are used to check whether a thread has finished its execution or not.

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

Ex: final boolean isAlive()

The **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

Ex: final void join() throws InterruptedException

Using **join()** method, we tell our 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.

Ex: final void join(long milliseconds) throws InterruptedException

As we have seen, the main thread must always be the last thread to finish its execution. Therefore, we can use Thread join() method to ensure that all the threads created by the program has been terminated before the execution of the main thread.

Java isAlive method:

Lets take an example and see how the isAlive() method works.

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

Output:

```
r1
true
true
r1
r2
r2
```

Example of thread with join() method:

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                Thread.sleep(500);
                }catch(InterruptedException ie){ }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
try{
                        t1.join();      //Waiting for t1 to finish
                }
catch(InterruptedException ie){}
                t2.start();
        }
}
```

Output:

```
r1
r2
r1
r2
```

In this above program join() method on thread t1 ensures that t1 finishes it process before thread t2 starts.

SYNCHRONIZATION

Synchronization of threads ensures that if two or more threads need to access a shared resource then that resource is used by only one thread at a time. We can synchronize our code using the ***synchronized*** keyword. We can invoke only one synchronized method for an object at any given time.

When a thread is within a synchronized method, all the other threads that try to call it on the same instance have to wait. During the execution of a synchronized method, the object is locked so that no other synchronized method can be invoked. The monitor is automatically released when the method completes its execution. The monitor can also be released when the synchronized method executes the wait ( ) method. When a thread calls the wait ( ) method, it temporarily releases the locks that it holds.

Synchronization among threads is achieved by using synchronized statements. The synchronized statement is used where the synchronization methods are not used in a class and we do not have access to the source code. We can synchronize the access to an object of this class by placing the calls to the methods defined by it inside a synchronized block. The general format is:

```
synchronized (obj) {
        // statements;
}
```

SUSPENDING & RESUMING THREADS

The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily stop execution. The suspended thread can be resumed using the **resume()** method.

Syntax

public final void suspend()

Example

```
public class JavaSuspendExp extends Thread
{
public void run()
{
  for(int i=1; i<5; i++)
  {
     try
     {
        // thread to sleep for 500 milliseconds
         sleep(500);
         System.out.println(Thread.currentThread().getName());
     }
     catch(InterruptedException e){System.out.println(e);}
```

```
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        // creating three threads
        JavaSuspendExp t1=new JavaSuspendExp ();
        JavaSuspendExp t2=new JavaSuspendExp ();
        JavaSuspendExp t3=new JavaSuspendExp ();
        // call run() method
        t1.start();
        t2.start();
        // suspend t2 thread
        t2.suspend();
        // call run() method
        t3.start();
    }
}
```

Output:

```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

COMMUNICATION BETWEEN THREADS (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

It causes the current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object.

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.

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 of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();
}
catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
}}
```

Output:
```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

Input/Output

Reading and Writing Data:

1. Reading and writing Characters

2. Reading and Writing Bytes

The subclasses of ***Reader*** and ***Writer*** implement streams that can handle characters. The two subclasses namely, ***FileReader*** and ***FileWriter*** are used to read and write characters from a file. These classes are used to read and write 16-bit characters.

Ex: Reading and Writing Characters using a file.

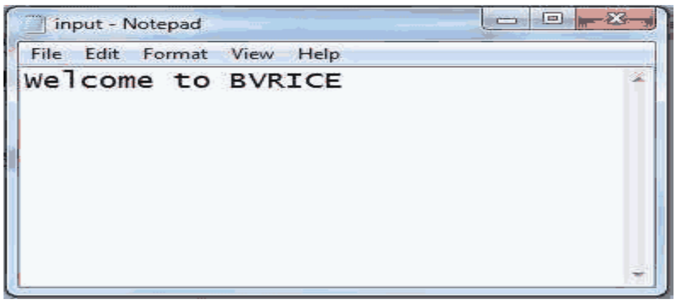
Step 1: *Open a new file in the Notepad and type the following program*.

```
import java.io.*;
class CopyCharacters {
        public static void main(String args[]) {
          // Declare and create input & output  files
          File infile=new File("input.dat");
          File outfile=new File("output.dat");
          FileReader ins=null; // Creates file stream ins
          FileWriter outs=null; // Create file stream outs
           try {
                ins=new FileReader(infile);
                outs=new FileWriter(outfile);
                // Read and write till the end;
                int ch;
                    while((ch=ins.read()) != -1) {
                      outs.write(ch);
                }
          }
          catch(IOException e) {
                System.out.println(e);
                System.exit(-1);
          }
          finally {       //close files
                try {
                      ins.close();
                      outs.close();
```

```
                        }
                        catch(IOException e) { }
            }
        }
  }
```

Step 2: Save the above program as " ***CopyCharacters.java***".

Step 3: Open a new file in the Notepad and type some string. *For example,*



```
Welcome to BVRICE
```

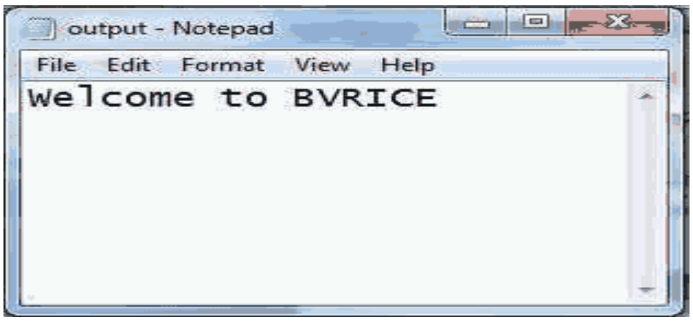
Save the above file as " ***input.dat***" in the same folder.

Step 4: Open a new file in the Notepad and don't type any data. Save the file as " ***output.dat***".

Step 5: Now ***compile*** the " ***CopyCharacters.java***" program using " ***javac***" command.

Step 6: If there is no more syntax errors, run the program using " ***java***" command.

Step 7: Open the " ***output.dat***" file and we can see the same string which we stored in the " ***input.dat***".



```
Welcome to BVRICE
```

Reading/Writing Bytes:

Most of the file systems use 8-bit bytes. Two commonly used classes for handling bytes are ***FileInputStream*** (*to read bytes from a file*) and ***FileOutputStream*** (*to write bytes to a file*) classes.

Ex: Copying bytes from one file to another (Read & Write).

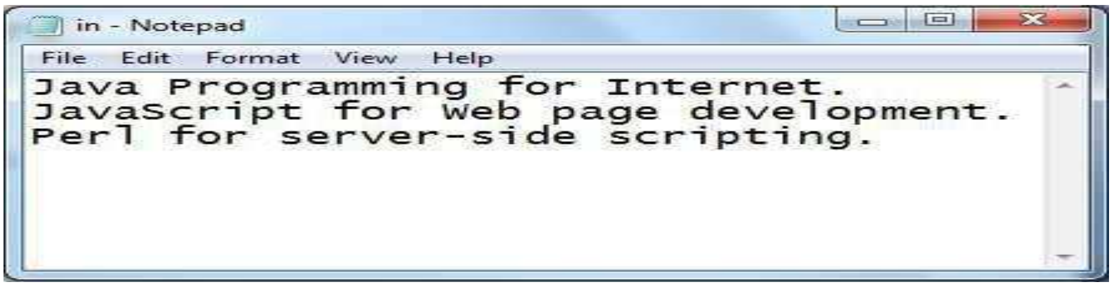
Step 1: Open a new file in Notepad and type the following code.

```
import java.io.*;
class CopyBytes {
   public static void main(String args[]) {
         FileInputStream infile=null;
```

```
            FileOutputStream outfile=null;
            byte br;
            try {
              infile=new FileInputStream("in.dat");
              outfile=new FileOutputStream("out.dat");
              do {
                  br=(byte)infile.read();
                     outfile.write(br);
              }
              while(br != -1);
            }
            catch(FileNotFoundException e) {
                     System.out.println("File Not Found");
            }
            catch(IOException e) {
                     System.out.println(e.getMessage());
            }
            finally {
                     try {
                              infile.close();
                              outfile.close();
                     }
                     catch(IOException e) { }
            }
       }
   }
```

Step 2: Save the above program as " ***CopyBytes.java***".

Step 3: Open a new file in Notepad and type some text as the following.



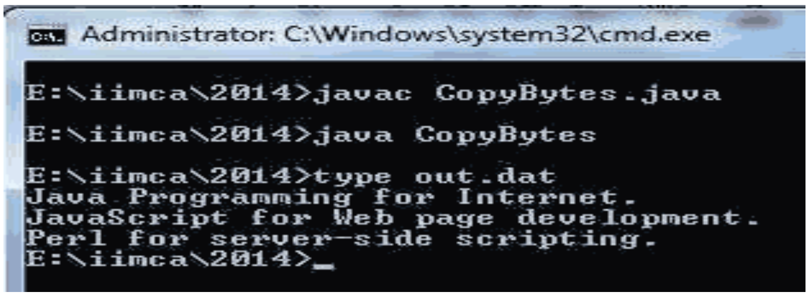
```
Java Programming for Internet.
JavaScript for Web page development.
Perl for server-side scripting.
```

and save the above file as " ***in.dat***".

Step 4: Open a new file in Notepad and save the file without data as " ***out.dat***".

Step 5: Compile and run the “CopyBytes.java” program.

Step 6: To see the output, use “ ***type***” command or open the “ ***out.dat***” in Notepad.



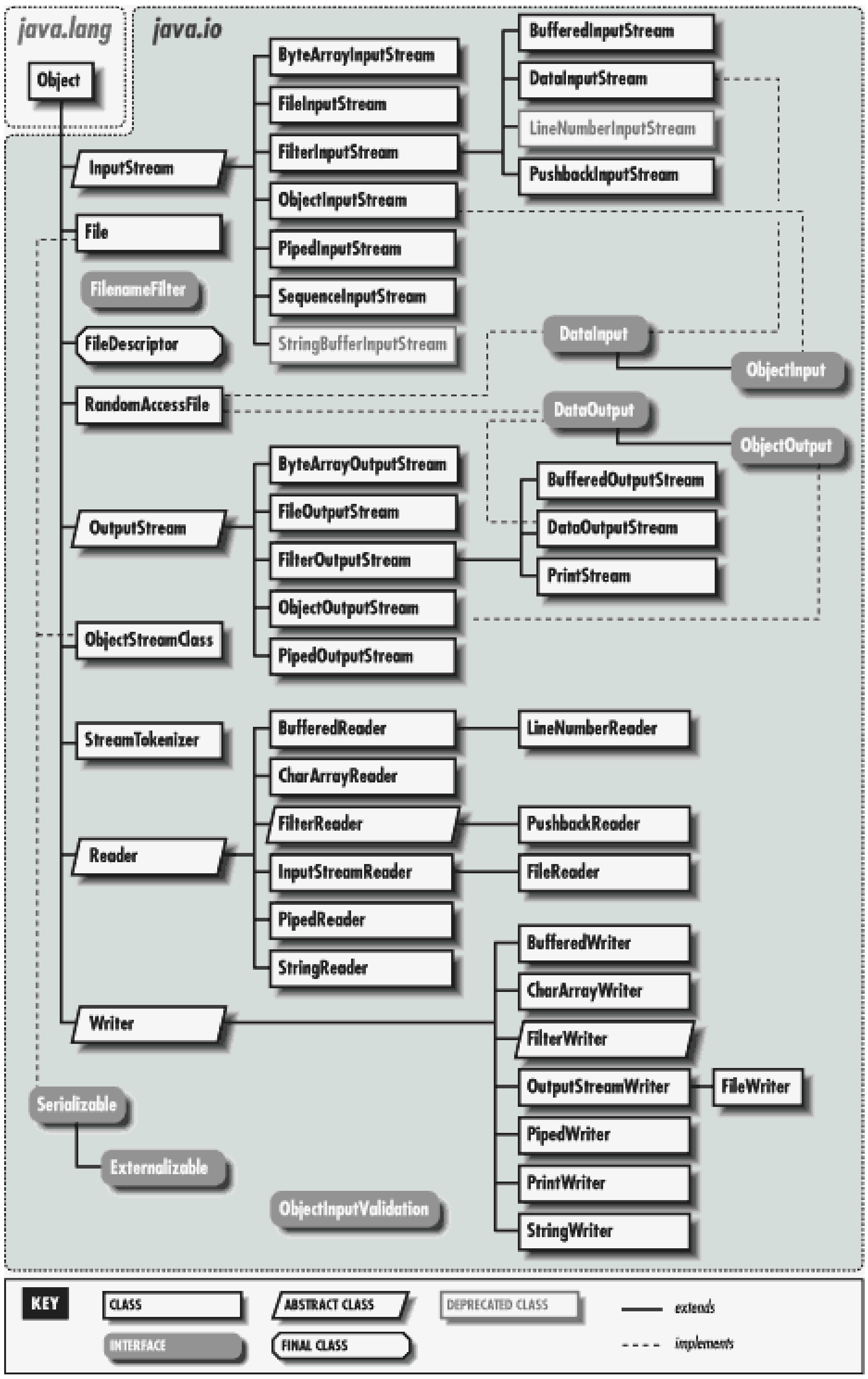
```
E:\iimca\2014>javac CopyBytes.java

E:\iimca\2014>java CopyBytes

E:\iimca\2014>type out.dat
Java Programming for Internet.
JavaScript for Web page development.
Perl for server-side scripting.
E:\iimca\2014>_
```

Java.io Package:

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.


java.lang
Object
java.io
InputStream
ByteArrayInputStream
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
File
FilenameFilter
FileDescriptor
RandomAccessFile
DataInput
ObjectInput
DataOutput
ObjectOutput
OutputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
ObjectStreamClass
StreamTokenizer
BufferedReader
LineNumberReader
CharArrayReader
FilterReader
PushbackReader
Reader
InputStreamReader
FileReader
PipedReader
StringReader
Writer
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
FileWriter
PipedWriter
PrintWriter
StringWriter
Serializable
Externalizable
ObjectInputValidation
KEY
CLASS
ABSTRACT CLASS
DEPRECATED CLASS
INTERFACE
FINAL CLASS
extends
implements

APPLETS

Applet:

An **Applet** is a small Internet-based program that has the Graphical User Interface (GUI), written in the Java programming language. Applets are designed to run inside a web browser or in applet viewer to facilitate the user to animate the graphics, play sound, and design the GUI components such as text box, button, and radio button. The applets are classified into two types. They are

a. Local Applet
b. Remote Applet

a. Local Applet:

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

b. Remote Applet:

A **remote applet** is that which is developed by someone else and stored on a remote computer connected to the internet.

If our system is connected to the internet then we can download it from remote computer and run it. In order to locate and load a remote applet, we must know the applet‘s address on the web. This address is known as Uniform Resource Locator (URL) and must be specified in applet‘s document.

The Applet Class:

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip

- Play an audio clip
- Resize the applet

Applet Structure:

The ***<Applet...>*** tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. The ***<Applet>*** tag requires minimum 3 (Three) attributes. They are

1. Name of the applet
2. Width of the applet (in pixels)
3. Height of the applet (in pixels)

Ex:

```
<Applet
        code = "Hellojava.class"
        width = 400
        Height = 200 >
</Applet>
```

We can execute the HTML file using "appletviewer" tool on the command line as the following: c:\ appletviewer RunApp.html

An example Applet Program:

How to run (execute) an Applet?

There are two ways to run an applet

1. By html file
2. By appletviewer tool (for testing purpose)

1. By html file: The following steps are required to execute an applet using html file.

Step 1: Open a new file in a text editor (for example, Notepad) and type the Java code.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
        g.drawString("welcome",150,150);
        }
  }
```

Note: class must be **public** because its object is created by Java Plugin software that resides on the browser.

Step 2: Save the file with **.java** extension in a folder (For example, “First.java”).

Step 3: Go to command prompt and compile it.

For example, C:\MCA> javac First.java

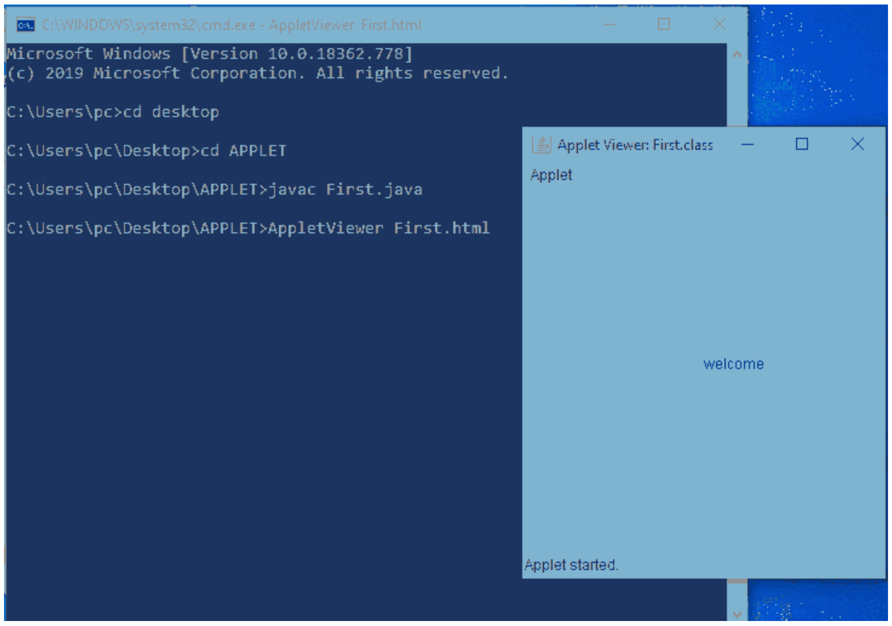
If there is no more syntax errors, Java compiler creates “First.class” file. Don’t execute with “java” interpreter.

Step 4: Open a new file in a text editor (for ex, Notepad) and type the html code.

```
<html>
    <body>
        <applet code="First.class" width="300" height="300">
        </applet>
    </body>
</html>
```

Step 5: Save the file with **.html** extension in a folder (For example, “First.html”).

Step 6: Go to command Prompt and execute it using appletviewer command



```
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\pc>cd desktop

C:\Users\pc\Desktop>cd APPLET

C:\Users\pc\Desktop\APPLET>javac First.java

C:\Users\pc\Desktop\APPLET>AppletViewer First.html
```

C:\WINDOWS\system32\cmd.exe - AppletViewer First.html
Applet Viewer: First.class
Applet
welcome
Applet started.

2. By appletviewer tool: The following steps are required to execute an applet in this method.

Step 1: Open a new file in a text editor (for example, Notepad) and type the Java code. The applet code is also written in comments within this file.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
  public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}
}
// <applet code="First.class" width="300" height="300">  </applet>
```

Step 2: Save the file with **.java** extension in a folder (For example, “First.java”).

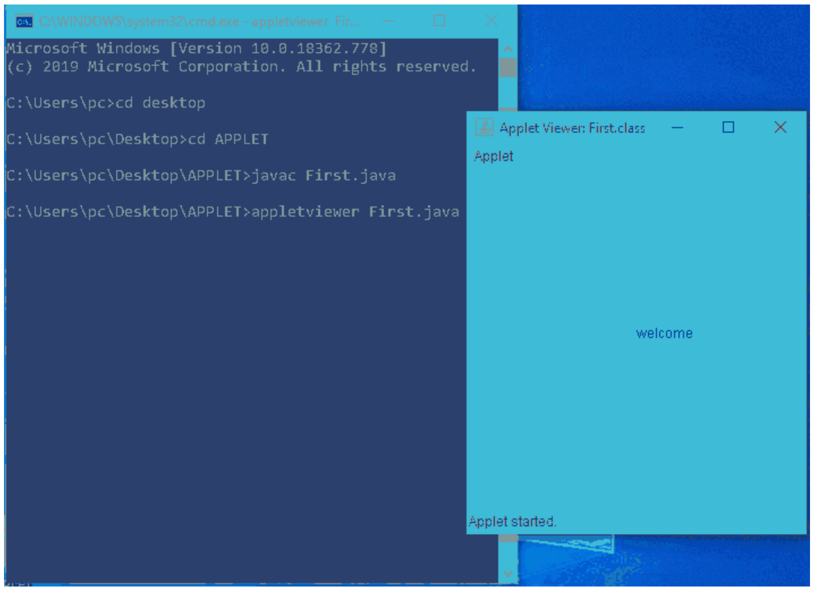
Step 3: Go to command prompt and compile it.

For example, C:\MCA> javac First.java

If there is no more syntax errors, Java compiler creates “First.class” file. Don’t execute with “java” interpreter.

Step 4: Now execute it using “appletviewer” tool.

For example, C:\MCA> appletviewer First.java


C:\WINDOWS\system32\cmd.exe - appletviewer Fir...
Applet Viewer: First.class
Applet
welcome
Applet started.

```
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\pc>cd desktop

C:\Users\pc\Desktop>cd APPLET

C:\Users\pc\Desktop\APPLET>javac First.java

C:\Users\pc\Desktop\APPLET>appletviewer First.java
```

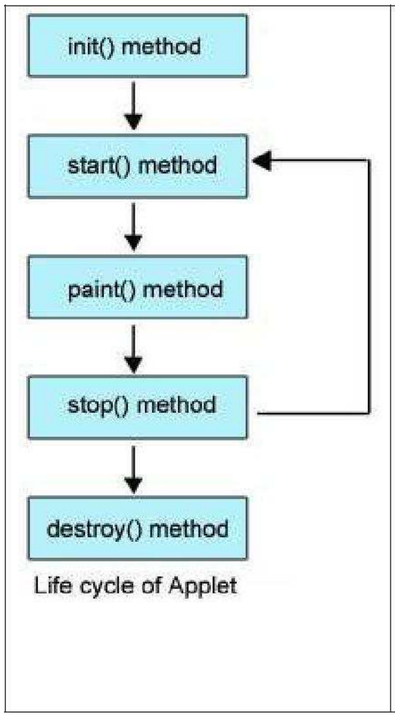
Applet Life Cycle:

Life Cycle of an Applet:

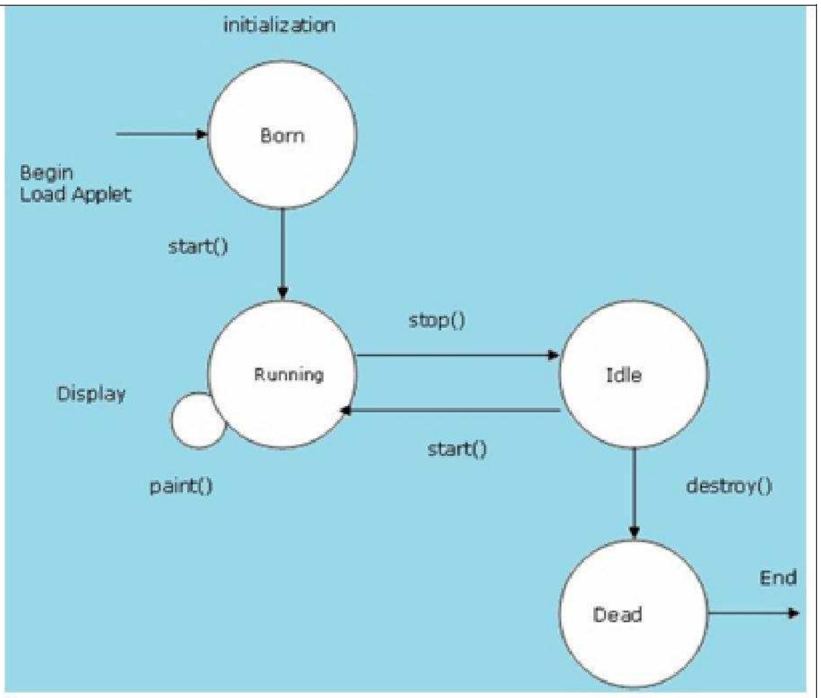
The life cycle of an applet consists the following phases:

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet: The ***java.applet.Applet*** class provides 4 life cycle methods and ***java.awt.Component*** class provides 1 (One) life cycle methods for an applet.



Life cycle of Applet



a. java.applet.Applet class:

For creating any applet ***java.applet.Applet*** class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.

3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

b. java.awt.Component class:

The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Paint(),update() and Repaint():

Paint(): The **paint ()** method supports painting via a **Graphics** object. This method holds instructions to paint this component.

Repaint(): The **repaint ()** method is used to cause paint () to be invoked by the **AWT** painting method. This method can't be overridden. You should call this method to get a component to repaint itself.

Update() : An update() method is called on calling the repaint method. The default implementation of the update() method clears the screen and calls the paint() method.This method is called in response to repaint() request.

Event Handling

Introduction

What is an Event?

Change in the state of an object is known as **event** i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as **event handler** that is executed when an event occurs. Java Uses the **Delegation Event Model** to handle the events. This model defines the standard mechanism to generate and handle the events. The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide classes for source object.
- **Listener** - It is also known as **event handler**. Listener is responsible for generating response to an event. Listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event and then returns.

Event deligation

Event handling code deals with events generated by GUI user interaction. The best practices for coding event handlers are event delegation model.

The event delegation model comprises three elements:

- Event source
- Event listener
- Adapter

Event source

An event source is a component, such as a GUI component, that generates an event. The event source can be generated by any type of user interaction.

In the event delegation model, a class represents each event type. Event objects are all defined in the java.util.Event Object subclasses.

Event listener

Event listeners are objects that receive notification of an event. When an event is fired, an event object is passed as an argument to the relevant listener object method. The listener object then handles the event.

All listeners are implementations of the EventListener interface or one of its subinterfaces. The Java API provides a predefined listener interface.

Adapter

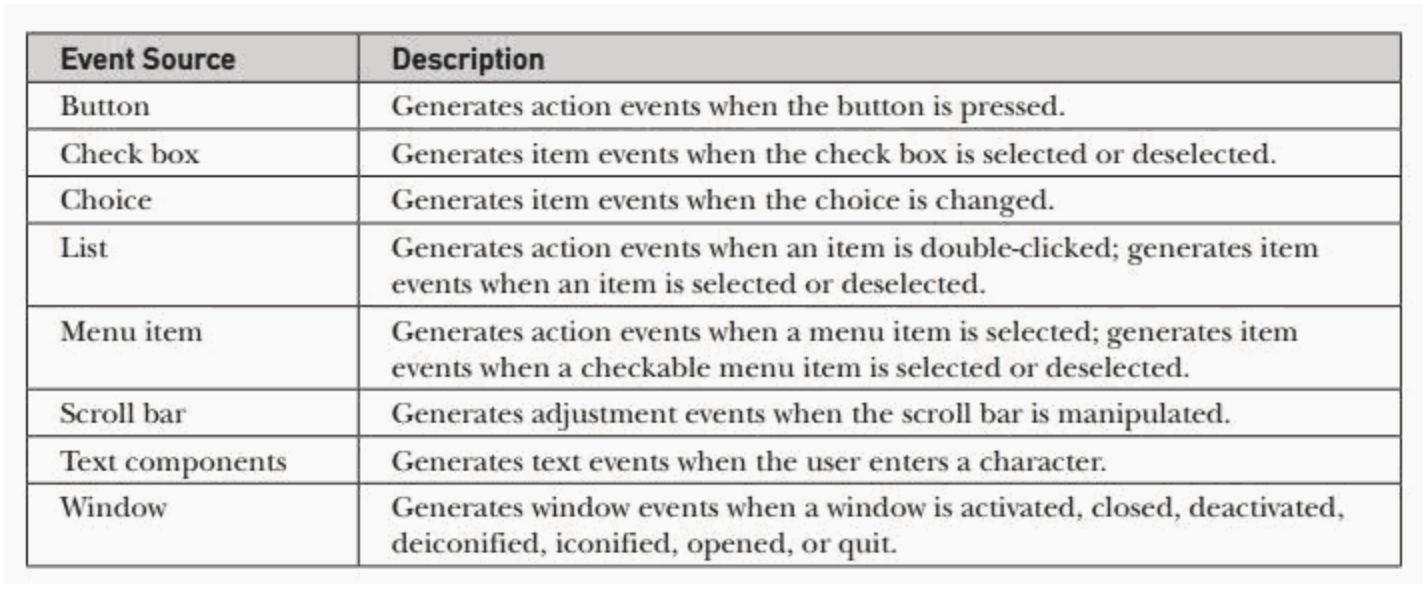
Adapters are abstract classes that implement listener interfaces using predefined methods. These are provided for convenience.

Java.awt.event description

It Provides interfaces and classes for dealing with different types of events fired by AWT components.

Interface	Description
ActionListener	The listener interface for receiving action events.
AdjustmentListener	The listener interface for receiving adjustment events.
AWTEventListener	The listener interface for receiving notification of events dispatched to objects that instances of Component or MenuComponent or their subclasses.
ComponentListener	The listener interface for receiving component events.
ContainerListener	The listener interface for receiving container events.
FocusListener	The listener interface for receiving keyboard focus events on a component.
HierarchyBoundsListener	The listener interface for receiving ancestor moved and resized events.
HierarchyListener	The listener interface for receiving hierarchy changed events.
InputMethodListener	The listener interface for receiving input method events.
ItemListener	The listener interface for receiving item events.
KeyListener	The listener interface for receiving keyboard events (keystrokes).
MouseListener	The listener interface for receiving "interesting" mouse events (press, release, click enter, and exit) on a component.
MouseMotionListener	The listener interface for receiving mouse motion events on a component.
MouseWheelListener	The listener interface for receiving mouse wheel events on a component.
TextListener	The listener interface for receiving text events.
WindowFocusListener	The listener interface for receiving WindowEvents, including WINDOW_GAINED_FOCUSand WINDOW_LOST_FOCUS events.
WindowListener	The listener interface for receiving window events.
WindowStateListener	The listener interface for receiving window state events.

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; generates item events when an item is selected or deselected.
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 Listerners

The Event listener represent the interfaces responsible to handle events. Java provides us various Event listener classes

AWT Event Listener Interfaces:

Following is the list of commonly used event listeners.

Sr. No.	**Control & Description**
1	ActionListener This interface is used for receiving the action events.
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.

4	KeyListener This interface is used for receiving the key events.
5	MouseListener This interface is used for receiving the mouse events.
6	TextListener This interface is used for receiving the text events.
7	WindowListener This interface is used for receiving the window events.
8	AdjustmentListener This interface is used for receiving the adjusmtent events.
9	ContainerListener This interface is used for receiving the container events.
10	MouseMotionListener This interface is used for receiving the mouse motion events.
11	FocusListener This interface is used for receiving the focus events

Adapter Classes

Adapter classes provide an implementation of listener interfaces. When you inherit the adapter class implementation for all methods is not mandatory. Thus writing excess code is saved.

These adapter classes can be found in **java.awt.event, java.awt.dnd** and **javax.swing.event** packages. Some of the common adapter classes with corresponding listener interfaces are given below.

- java.awt.event
- java.awt.dnd
- javax.swing.event

java.awt.event Adapter classes:

Adapter class	**Listener interface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes:

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes:

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Inner Classes

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) Nested Inner class

2) Method Local inner classes

3) Anonymous inner classes

4) Static nested classes

1) Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.

Like class, interface can also be nested and can have access specifiers.

Following example demonstrates a nested class.

```
class Outer {
   // Simple nested inner class
   class Inner {
      public void show() {
           System.out.println("In a nested class method");
      }
   }
}
class Main {
   public static void main(String[] args) {
       Outer.Inner in = new Outer().new Inner();
       in.show();
   }
}
```

Output:

```
In a nested class method
```

2) Method Local inner classes

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
```

```
class MethodDemo {
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Output

```
Inside outerMethod
Inside innerMethod
```

3) Static nested classes

Static nested classes are not technically an inner class. They are like a static member of outer class.

```
class Outer {
   private static void outerMethod() {
     System.out.println("inside outerMethod");
   }

   // A static inner class
   static class Inner {
     public static void main(String[] args) {
        System.out.println("inside inner class Method");
        outerMethod();
     }
   }

}
```

Output

```
inside inner class Method
inside outerMethod
```

4) Anonymous inner classes

Anonymous inner classes are declared without any name at all. They are created

As subclass of specified type

Ex:

```
class Demo {
   void show() {
      System.out.println("i am in show method of super class");
   }
}
class Flavor1Demo {
 //  An anonymous class with Demo as base class
   static Demo d = new Demo() {
       void show() {
           super.show();
           System.out.println("i am in Flavor1Demo class");
       }
   };
   public static void main(String[] args){
       d.show();
   }
}
```

Output

```
i am in show method of super class
i am in Flavor1Demo class
```

AWT

Why AWT?

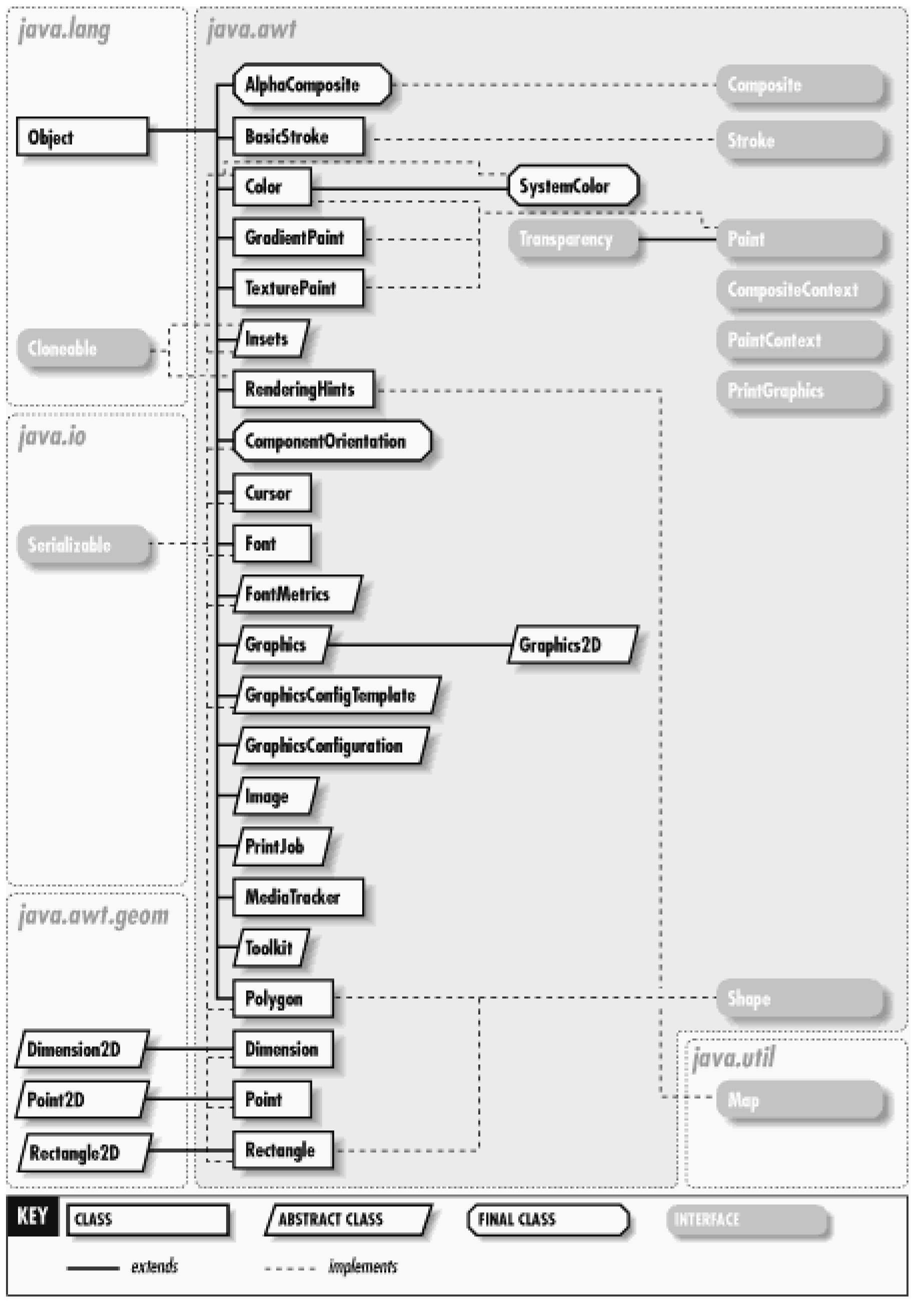
Java AWT is also known as Abstract Window Toolkit is an API that is used to develop either GUI or window-based applications in Java. Java AWT components are platform-dependent which implies that they are displayed according to the view of the operating system. It is also heavyweight implying that its components are using the resources of the Operating System. java.Awt package provides classes for AWT api.

For example, TextField, CheckBox, Choice, Label, TextArea, Radio Button, List, etc.

Java.awt.Package

The java.awt package is the main package of the AWT, or Abstract Windowing Toolkit. It contains classes for graphics, including the Java 2D graphics capabilities, and also defines the basic graphical user interface (GUI) framework for Java.

java.awt also includes a number of heavyweight GUI objects, many of which have been defined by the javax.swing package. java.awt also has a number of important subpackages. The most important graphics classes in java.awt are Graphics and its Java 2D extension, Graphics2D. These classes represent a drawing surface, maintain a set of drawing attributes, and define methods for drawing and filling lines, shapes, and text. Classes that represent graphics attributes include Color, Font, Paint, Stroke, and Composite. The following Figure shows the graphics classes of this package.



Components and containers

A *component* is the fundamental user interface object in Java. Everything you see on the display in a Java application is a component. This includes things like windows, panels, buttons, checkboxes, scrollbars, lists, menus, and text fields. To be used, a component usually must be placed in a *container*.

The Container is one of the components in AWT that contains other components like buttons, text fields, labels, etc. The classes that extend Container class are known as containers such as Frame, Dialog, and Panel as shown in the hierarchy.

Types of containers:

As demonstrated above, container refers to the location where components can be added like text field, button, checkbox, etc. There are in total, four types of containers available in AWT, that is, **Window, Frame, Dialog, and Panel**. As shown in the hierarchy above, Frame and Dialog are subclasses of Window class.

Window

The window is a container which does not have borders and menu bars. In order to create a window, you can use frame, dialog or another window.

Panel

The Panel is the container/class that doesn't contain the title bar and menu bars. It has other components like button, text field, etc.

Dialog

The Dialog is the container or class having border and title. We cannot create an instance of the Dialog class without an associated instance of the respective Frame class.

Button

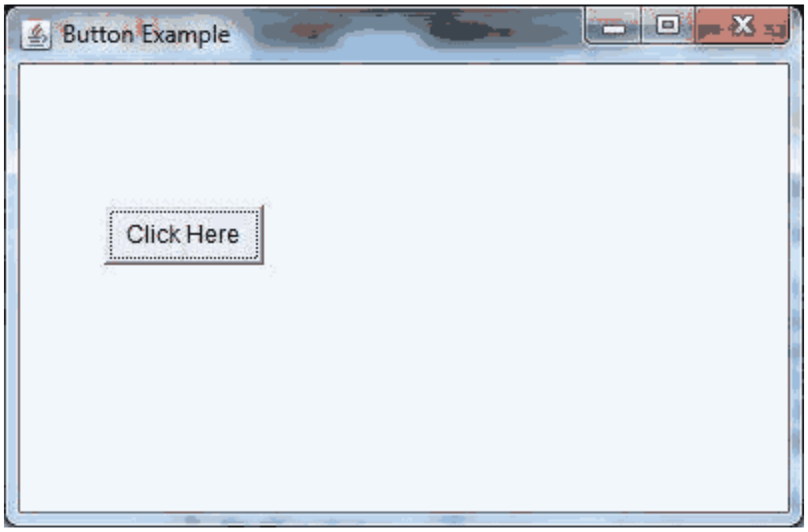
The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

Sy: **public class** Button **extends** Component **implements** Accessible

Ex:

```
 import java.awt.*;
public class ButtonExample {
public static void main(String[] args) {
   Frame f=new Frame("Button Example");
   Button b=new Button("Click Here");
   b.setBounds(50,100,80,30);
   f.add(b);
   f.setSize(400,400);
```

```
    f.setLayout(null);
    f.setVisible(true);
}
}
```



Label

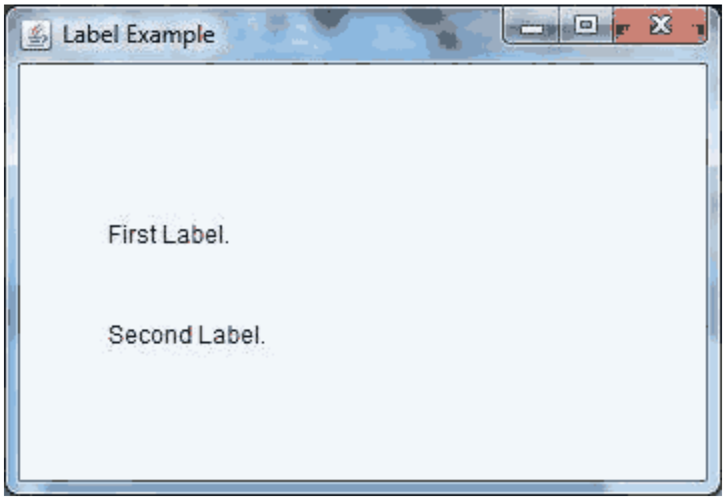
The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

Sy: **public class** Label **extends** Component **implements** Accessible

Ex:

```
 import java.awt.*;
class LabelExample{
public static void main(String args[]){
    Frame f= new Frame("Label Example");
    Label l1,l2;
    l1=new Label("First Label.");
    l1.setBounds(50,100, 100,30);
    l2=new Label("Second Label.");
    l2.setBounds(50,150, 100,30);
    f.add(l1); f.add(l2);
```

```
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}
}
```



First Label.

Second Label.

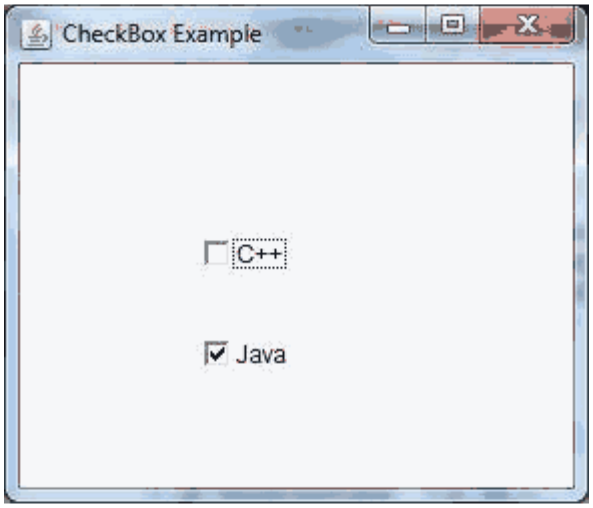
Checkbox

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

Sy: public class Checkbox **extends** Component **implements** ItemSelectable, Accessible

Ex:

```
import java.awt.*;
public class CheckboxExample
{
     CheckboxExample(){
       Frame f= new Frame("Checkbox Example");
        Checkbox checkbox1 = new Checkbox("C++");
        checkbox1.setBounds(100,100, 50,50);
        Checkbox checkbox2 = new Checkbox("Java", true);
        checkbox2.setBounds(100,150, 50,50);
        f.add(checkbox1);
        f.add(checkbox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
    new CheckboxExample();
}
}
```



Radio Button

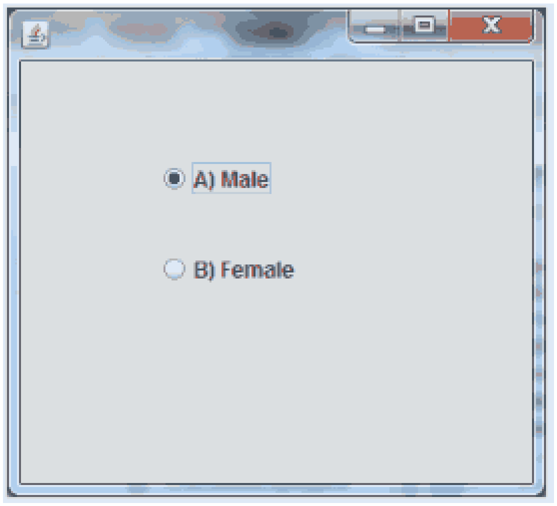
The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

Sy: public class JRadioButton **extends** JToggleButton **implements** Accessible

Ex:

```
import javax.swing.*;
public class RadioButtonExample {
JFrame f;
RadioButtonExample(){
f=new JFrame();
JRadioButton r1=new JRadioButton("A) Male");
JRadioButton r2=new JRadioButton("B) Female");
r1.setBounds(75,50,100,30);
r2.setBounds(75,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(r1);bg.add(r2);
f.add(r1);f.add(r2);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String[] args) {
   new RadioButtonExample();
}
}
```



List Boxes

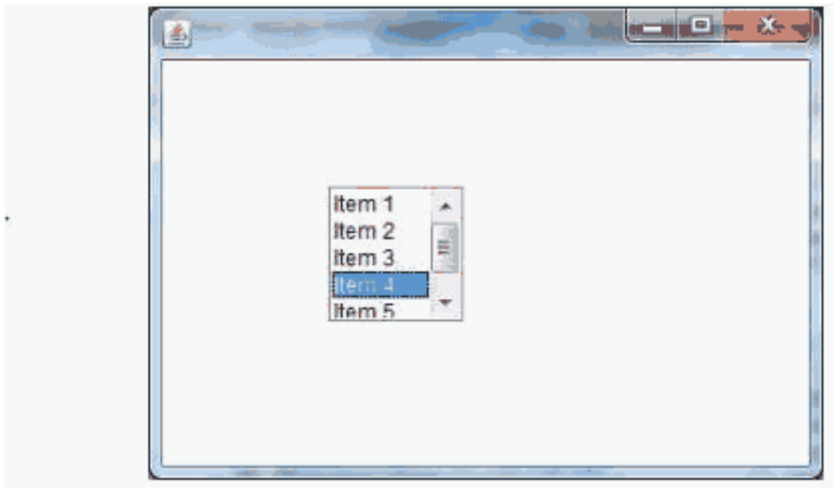
The object of List class represents a list of text items. By the help of list, user can choose either one item or multiple items. It inherits Component class.

Sy: **public class** List **extends** Component **implements** ItemSelectable, Accessible

Ex:

```
import java.awt.*;
public class ListExample
```

```
{
    ListExample(){
      Frame f= new Frame();
      List l1=new List(5);
      l1.setBounds(100,100, 75,75);
      l1.add("Item 1");
      l1.add("Item 2");
      l1.add("Item 3");
      l1.add("Item 4");
      l1.add("Item 5");
      f.add(l1);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
{
  new ListExample();
}
}
```



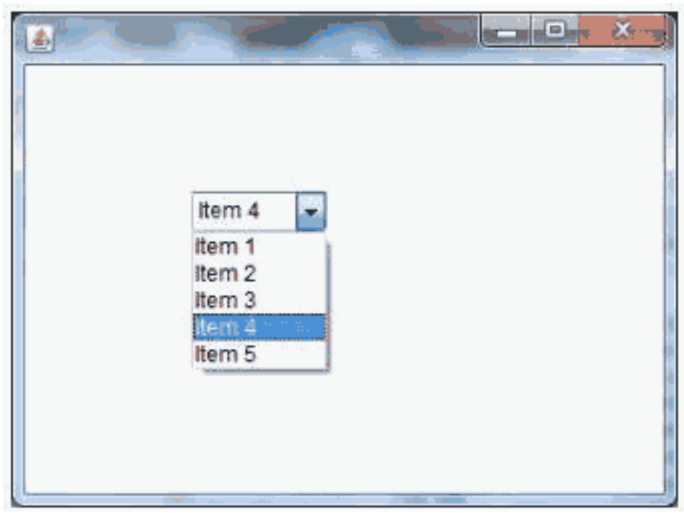
Choice Boxes

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

Sy: **public class** Choice **extends** Component **implements** ItemSelectable, Accessible

Ex:

```
import java.awt.*;
public class ChoiceExample
{
        ChoiceExample(){
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(100,100, 75,75);
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
        f.add(c);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
      }
public static void main(String args[])
{
   new ChoiceExample();
}
}
```



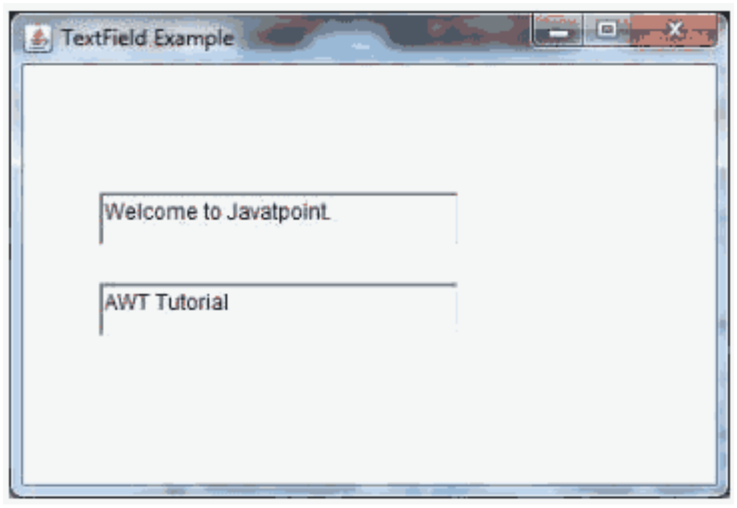
Text Field

The object of a TextField class is a text component that allows the editing of a single line text. It inherits TextComponent class.

Sy: **public class** TextField **extends** TextComponent

Ex:

```
import java.awt.*;
class TextFieldExample{
public static void main(String args[]){
    Frame f= new Frame("TextField Example");
    TextField t1,t2;
    t1=new TextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new TextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```



Welcome to Javatpoint

AWT Tutorial

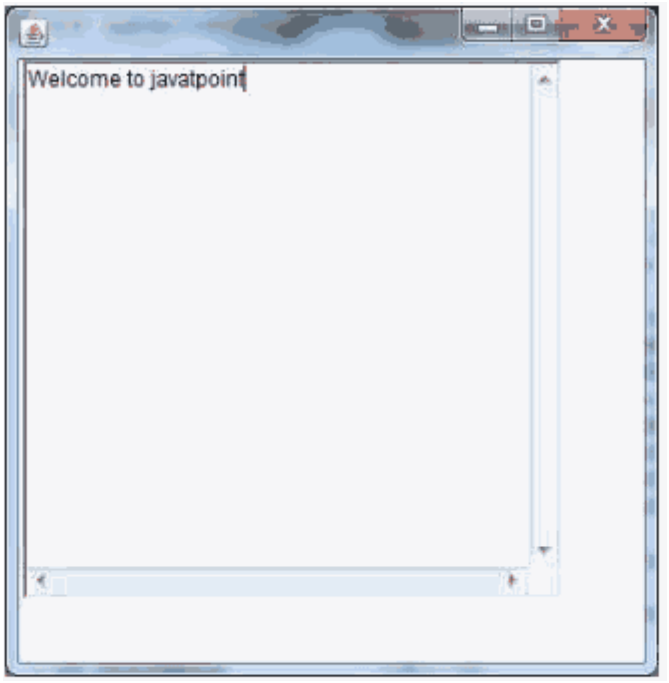
TextArea:

The object of a TextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

Sy: **public class** TextArea **extends** TextComponent

Ex:

```
import java.awt.*;
public class TextAreaExample
{
    TextAreaExample(){
       Frame f= new Frame();
           TextArea area=new TextArea("Welcome to bvrice ");
       area.setBounds(10,30, 300,300);
       f.add(area);
       f.setSize(400,400);
       f.setLayout(null);
       f.setVisible(true);
    }
public static void main(String args[])
{
  new TextAreaExample();
}
}
```


Welcome to javatpoint

Container classes

The Component class is found under java.awt package. The container class is the subclass of Component class. The Component class defines a number of methods for handling events, changing window bounds, controlling fonts and colors, and drawing components and their content.

A container is a component that can accommodate other components and also other containers. Containers provide the support for building complex hierarchical graphical user interface. Container provides the overloaded method add () to include components in the container.

Layouts

The java.awt library provides 5 basic layouts. Each layout has its own significance and all of them are completely different. Let us learn how to apply any layout to a frame or a panel and also discuss about each layout in brief.

The 5 layouts available in the java.awt library are:

1. Border Layout
2. Grid Layout
3. GridBag Layout
4. Card Layout
5. Flow Layout

1. Border Layout

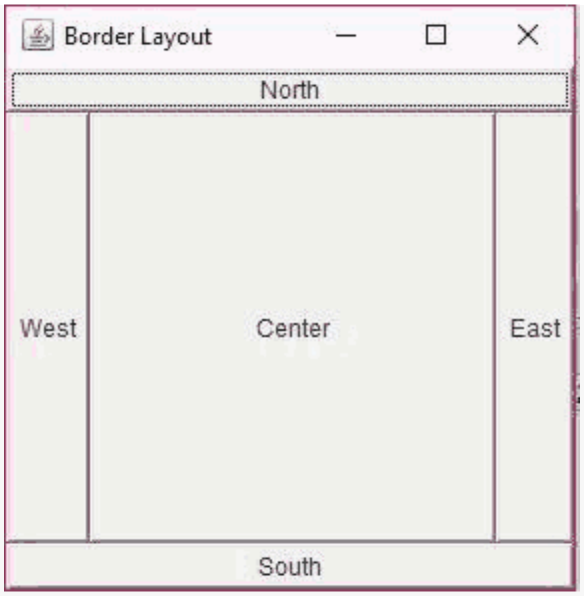
The BorderLayout is a layout which organizes components in terms of direction. A border layout divides the frame or panel into 5 sections – North, South, East, West and Centre. Each component can be arranged in a particular direction by passing an additional argument.

Sy: BorderLayoutExample.java

Ex:

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Label;
import javax.swing.JFrame;
public class BorderLayoutExample extends JFrame {
   public static void main(String[] args) {
      BorderLayoutExample a = new BorderLayoutExample();
   }
   public BorderLayoutExample() {
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      BorderLayout b = new BorderLayout();
      setTitle("Border Layout");
      setSize(300, 300);
      add(new Button("North"), BorderLayout.NORTH);
      add(new Button("South"), BorderLayout.SOUTH);
      add(new Button("East"), BorderLayout.EAST);
      add(new Button("West"), BorderLayout.WEST);
      add(new Button("Center"), BorderLayout.CENTER);
   }
}
```

Output



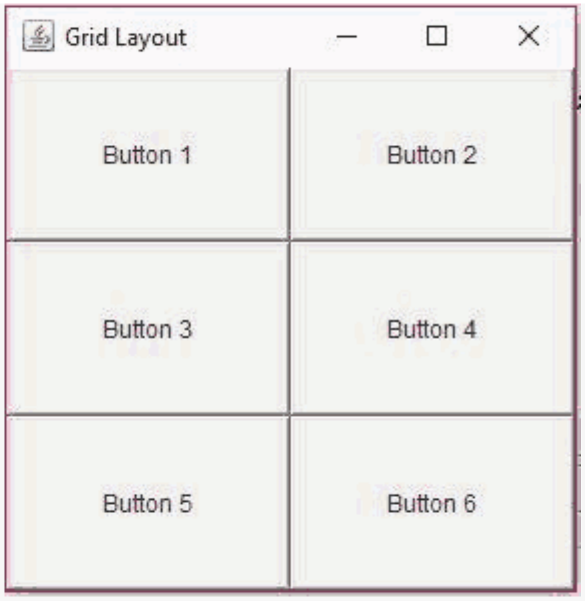
2. Grid Layout

A GridLayout is a more organized way of arranging components. It divides the frame or panel in the form of a grid containing evenly distributed cells. Each component get added to a particular cell. The order of placement of components is directly dependant on the order in which they are added to the frame or panel. The below image shows a 2 column 3 row GridLayout based Frame.

The arguments of the constructor GridLayout(int row,int cols) decide the grid size.

```
GridLayoutExample.java
import java.awt.Button;
import java.awt.GridLayout;
import javax.swing.JFrame;
public class GridLayoutExample extends JFrame {
    public static void main(String[] args) {
        GridLayoutExample a = new GridLayoutExample();
    }
    public GridLayoutExample() {
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        GridLayout g = new GridLayout(3, 2);
        setLayout(g);
        setTitle("Grid Layout");
        setSize(300, 300);
        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));
        add(new Button("Button 6"));
    }
}
```

Output



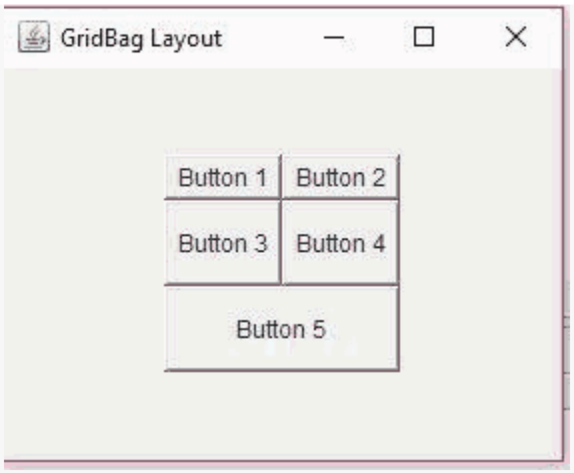
3. GridBag Layout

The GridBagLayout is the most flexible layout which provides an organized yet flexible way to arrange components. It provides developers with the flexibility to choose the exact location of the component, in a grid, it’s row span and column span as well as the horizontal and vertical gap. The below image shows a GridBagLayout. It contains Button 5 which spans 2 rows at a time.

GridBagLayoutExample.java

```
import java.awt.Button;
import java.awt.CardLayout;
import java.awt.FlowLayout;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JFrame;
public class GridBagLayoutExample extends JFrame {
    public static void main(String[] args) {
        GridBagLayoutExample a = new GridBagLayoutExample();
    }
    public GridBagLayoutExample() {
        setSize(300, 300);
        setPreferredSize(getSize());
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        GridBagLayout g = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(g);
        setTitle("GridBag Layout");
        GridBagLayout layout = new GridBagLayout();
        this.setLayout(layout);
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.gridx = 0;
        gbc.gridy = 0;
        this.add(new Button("Button 1"), gbc);
        gbc.gridx = 1;
        gbc.gridy = 0;
        this.add(new Button("Button 2"), gbc);
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.ipady = 20;
        gbc.gridx = 0;
        gbc.gridy = 1;
        this.add(new Button("Button 3"), gbc);
        gbc.gridx = 1;
        gbc.gridy = 1;
        this.add(new Button("Button 4"), gbc);
        gbc.gridx = 0;
        gbc.gridy = 2;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.gridwidth = 2;
        this.add(new Button("Button 5"), gbc);
    }
}
```

Output



4. Card Layout

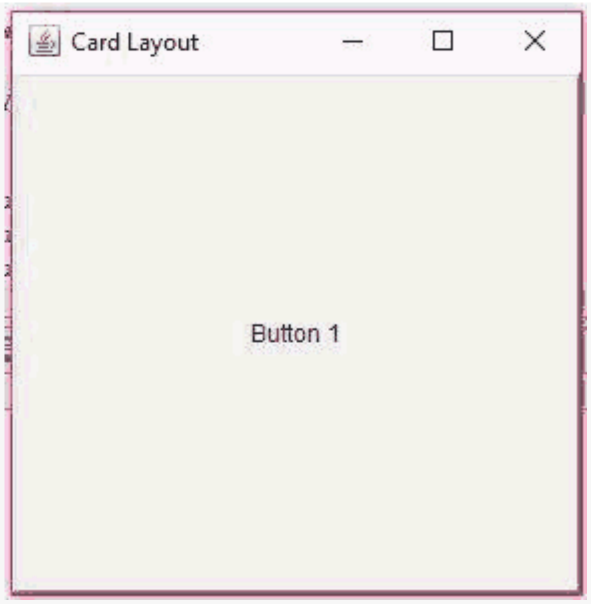
This is the layout which is rarely used and is utilized to stack up components one above another. The CardLayout allows components to stay over one another and switch any component to the front as per the requirement. let us understand it with a small example. Consider the code below:

```
mychoice.addItemListener(new ItemListener() {
         public void itemStateChanged(ItemEvent e) {
            CardLayout cardLayout = (CardLayout)(e.getTarget().getParent().getLayout());
            cardLayout.show(panel, (String)e.getItem());
        }
    });
```

The above code is associated with as an event listener with a combo box. As per the change in value of the combo, the component gets shown. In order to create a card layout, you can use the below code

```
CardLayoutExample.java
import java.awt.Button;
import java.awt.CardLayout;
import javax.swing.JFrame;
public class CardLayoutExample extends JFrame {
    public static void main(String[] args) {
        CardLayoutExample a = new CardLayoutExample();
    }
    public CardLayoutExample() {
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        CardLayout g = new CardLayout();
        setLayout(g);
        setTitle("Card Layout");
        setSize(300, 300);
        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));
        add(new Button("Button 6"));
    }
}
```

Output



5. Flow Layout

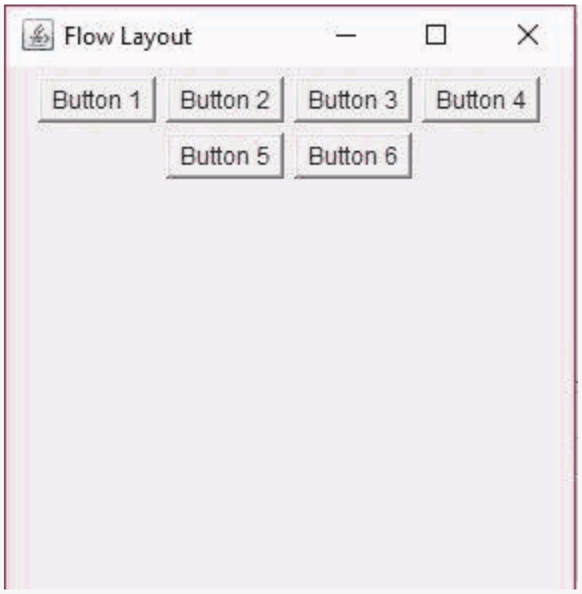
As the name indicates, FlowLayout is the layout which allows components to flow to and end of the visible part is reached. A FlowLayout basically helps develop more responsive UI and keep the components in a free flowing manner. The below image shows an actual flow layout with 6 components.

Since this is the default layout for a frame or panel, it can also work without applying the layout explicitly.

```
FlowLayoutExample.java
import java.awt.Button;
import java.awt.FlowLayout;
import javax.swing.JFrame;
public class FlowLayoutExample extends JFrame {
   public static void main(String[] args) {
      FlowLayoutExample a = new FlowLayoutExample();
   }
   public FlowLayoutExample() {
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      FlowLayout g = new FlowLayout();
```

```
        setLayout(g);
        setTitle("Flow Layout");
        setSize(300, 300);
        add(new Button("Button 1"));
        add(new Button("Button 2"));
        add(new Button("Button 3"));
        add(new Button("Button 4"));
        add(new Button("Button 5"));
        add(new Button("Button 6"));
    }
}
```

Output



Menu

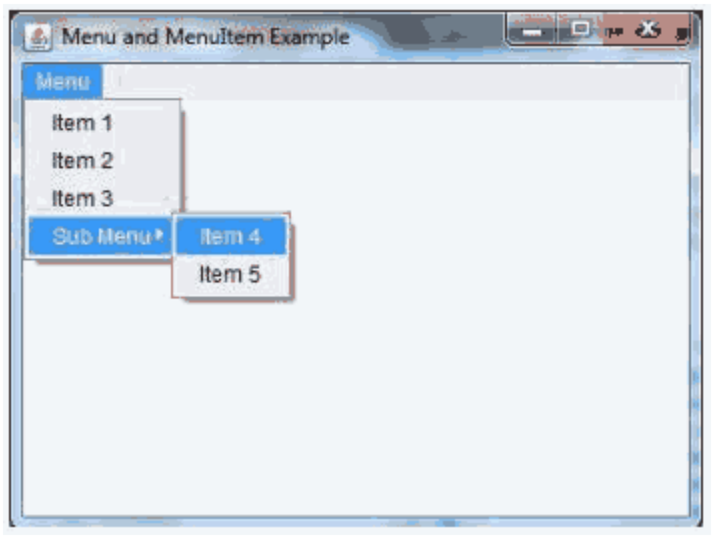
The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

Sy: **public class** Menu **extends** MenuItem **implements** MenuContainer, Accessible

Ex:

import java.awt.*;

```
class MenuExample
{
    MenuExample(){
        Frame f= new Frame("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Sub Menu");
        MenuItem i1=new MenuItem("Item 1");
        MenuItem i2=new MenuItem("Item 2");
        MenuItem i3=new MenuItem("Item 3");
        MenuItem i4=new MenuItem("Item 4");
        MenuItem i5=new MenuItem("Item 5");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```



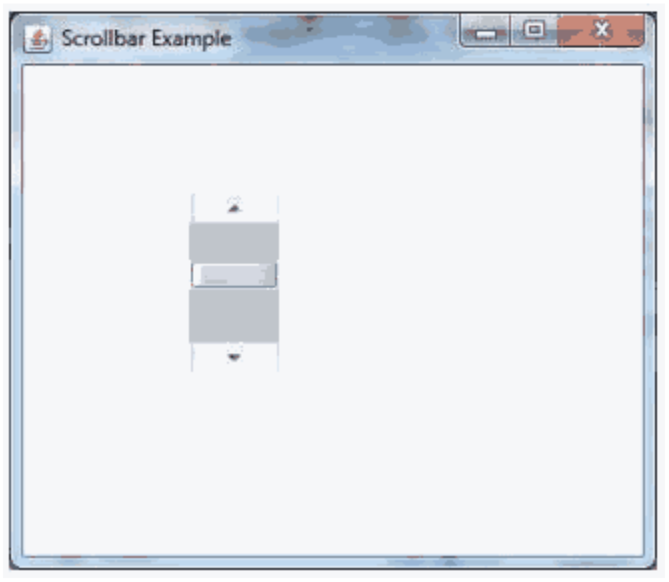
ScrollBar

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

Sy: **public class** Scrollbar **extends** Component **implements** Adjustable, Accessible

Ex:

```
import java.awt.*;
class ScrollbarExample{
ScrollbarExample(){
        Frame f= new Frame("Scrollbar Example");
        Scrollbar s=new Scrollbar();
        s.setBounds(100,100, 50,100);
        f.add(s);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[]){
     new ScrollbarExample();
}
}
```


Scrollbar Example

SWINGS

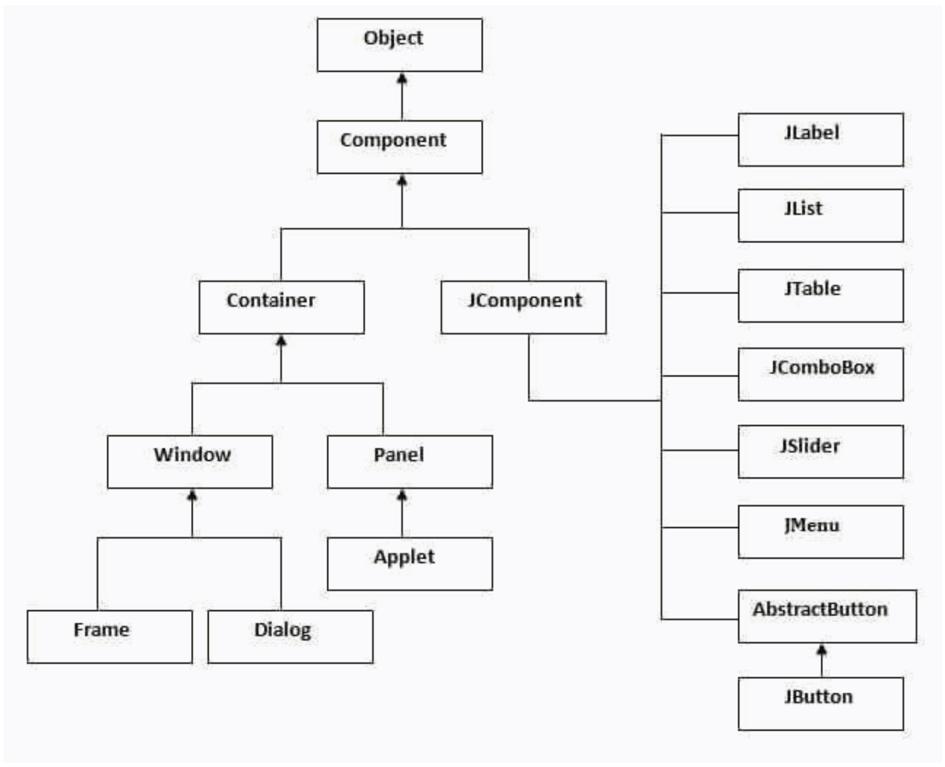
Introduction: Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The **javax.swing package** provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



.

Example swing prog:

import javax.swing.*;

public class FirstSwingExample {

public static void main(String[] args) {

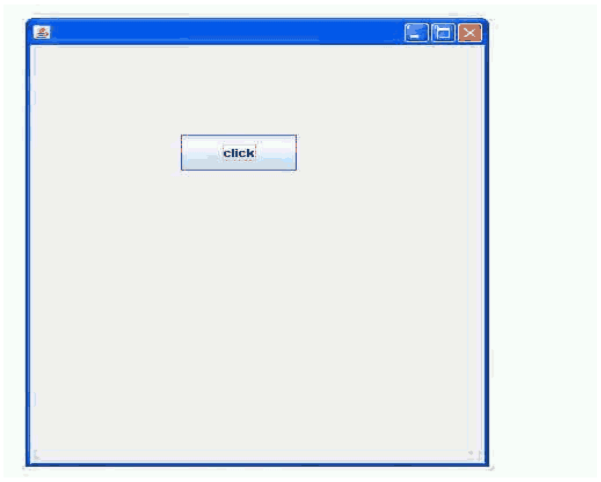
JFrame f=**new** JFrame();//creating instance of JFrame

JButton b=**new** JButton("click");//creating instance of JButton

b.setBounds(130,100,100, 40);//x axis, y axis, width, height

f.add(b);//adding button in JFrame

```
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
```



```
}
```

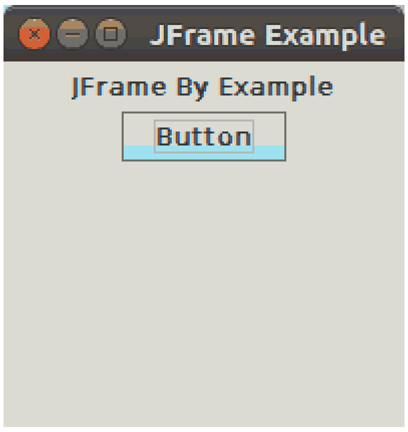
JFrame:

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

Ex:

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.Jpanel;
public class JFrameExample {
    public static void main(String s[]) {
        JFrame frame = new JFrame("JFrame Example");
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());
        JLabel label = new JLabel("JFrame By Example");
        JButton button = new JButton();
        button.setText("Button");
        panel.add(label);
        panel.add(button);
        frame.add(panel);
        frame.setSize(200, 300);
        frame.setLocationRelativeTo(null);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```



JApplet:

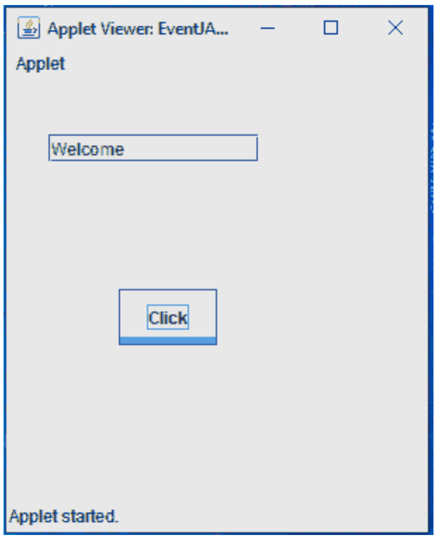
As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class.

Ex:

```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
JButton b;
JTextField tf;
public void init(){
tf=new JTextField();
tf.setBounds(30,40,150,20);
b=new JButton("Click");
b.setBounds(80,150,70,40);
add(b);add(tf);
b.addActionListener(this);
setLayout(null);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
}
```

In the above example, we have created all the controls in init() method because it is invoked only once.

```
<html>
<body>
<applet code="EventJApplet.class" width="300" height="300">
</applet>
</body>
</html>
```



JPanel: The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

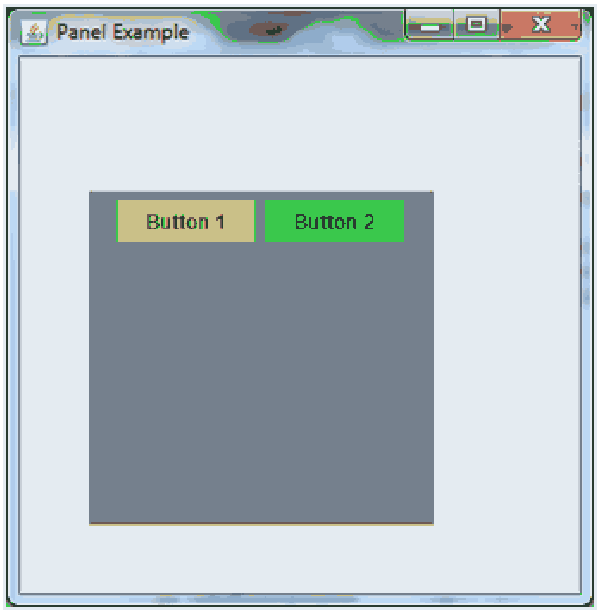
It doesn't have title bar.

Sy: public class JPanel **extends** JComponent **implements** Accessible

Ex:

```
import java.awt.*;
import javax.swing.*;
public class PanelExample {
    PanelExample()
      {
```

```
    JFrame f= new JFrame("Panel Example");
    JPanel panel=new JPanel();
    panel.setBounds(40,80,200,200);
    panel.setBackground(Color.gray);
    JButton b1=new JButton("Button 1");
    b1.setBounds(50,100,80,30);
    b1.setBackground(Color.yellow);
    JButton b2=new JButton("Button 2");
    b2.setBounds(100,100,80,30);
    b2.setBackground(Color.green);
    panel.add(b1); panel.add(b2);
    f.add(panel);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
    }
    public static void main(String args[])
    {
    new PanelExample();
    }
}
```



Components in Swings:

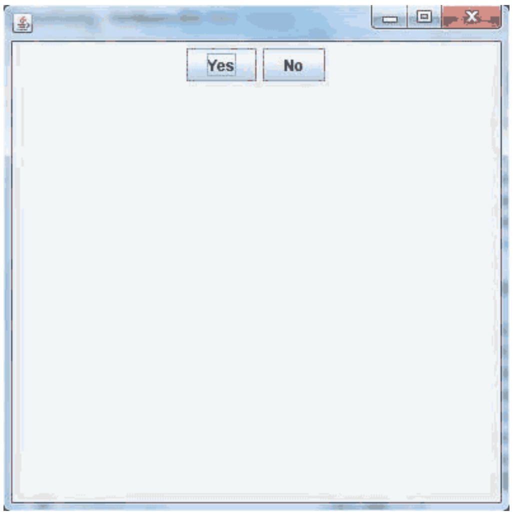
Swing Framework contains a large set of components which provide rich functionalities and allow high level of customization. All these components are lightweight components. They all are derived from **JComponent** class. It supports the pluggable look and feel.

1. JButton:

JButton class provides functionality of a button. JButton supports **ActionEvent**. When a button is pressed an **ActionEvent** is generated.

Example using JButton

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class testswing extends JFrame
{
 testswing()
 {
  JButton bt1 = new JButton("Yes");   //Creating a Yes Button.
  JButton bt2 = new JButton("No");    //Creating a No Button.
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)     //setting close operation.
  setLayout(new FlowLayout()); //setting layout using FlowLayout object
  setSize(400, 400); //setting size of Jframe
  add(bt1); //adding Yes button to frame.
  add(bt2); //adding No button to frame.
  setVisible(true);
 }
 public static void main(String[] args)
 {
  new testswing();
 }
}
```

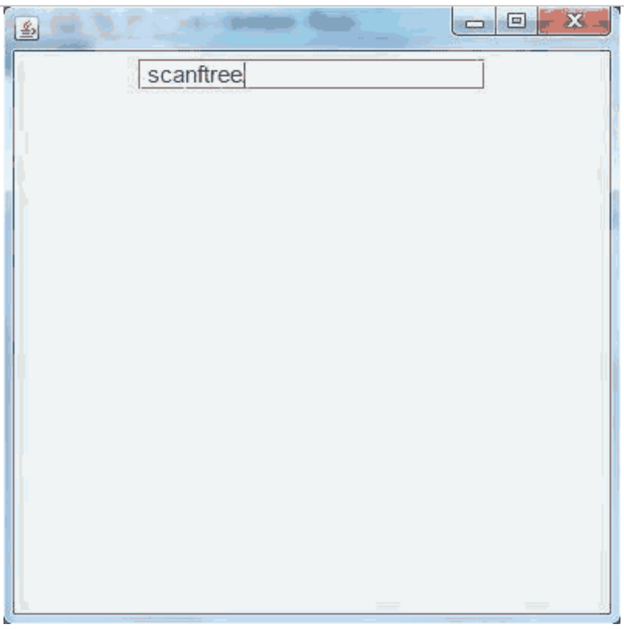


JTextField:

JTextField is used for taking input of single line of text. It is most widely used text component.

Example using JTextField

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class MyTextField extends JFrame
{
 public MyTextField()
 {
  JTextField jtf = new JTextField(20);//creating JTextField.
  add(jtf);//adding JTextField to frame.
  setLayout(new FlowLayout());
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(400, 400);
  setVisible(true);
 }
 public static void main(String[] args)
 {
  new MyTextField();
 }
}
```

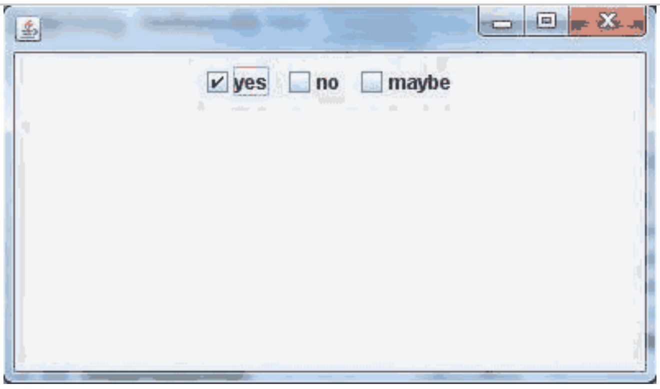


JCheckBox:

JCheckBox class is used to create checkboxes in frame.

Example using JCheckBox

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Test extends JFrame
{
 public Test()
 {
  JCheckBox jcb = new JCheckBox("yes");   //creating JCheckBox.
  add(jcb);                               //adding JCheckBox to frame.
  jcb = new JCheckBox("no");              //creating JCheckBox.
  add(jcb);                               //adding JCheckBox to frame.
  jcb = new JCheckBox("maybe");           //creating JCheckBox.
  add(jcb);                               //adding JCheckBox to frame.
  setLayout(new FlowLayout());
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(400, 400);
  setVisible(true);
 }
 public static void main(String[] args)
 {
  new Test();
 }
}
```

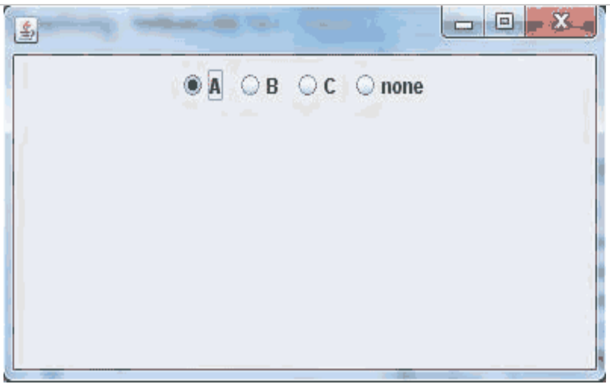


JRadioButton:

Radio button is a group of related button in which only one can be selected. JRadioButton class is used to create a radio button in Frames.

Example using JRadioButton

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Test extends JFrame
{
 public Test()
 {
  JRadioButton jcb = new JRadioButton("A");//creating JRadioButton.
  add(jcb);//adding JRadioButton to frame.
  jcb = new JRadioButton("B");//creating JRadioButton.
  add(jcb);//adding JRadioButton to frame.
  jcb = new JRadioButton("C");//creating JRadioButton.
  add(jcb);//adding JRadioButton to frame.
  jcb = new JRadioButton("none");
  add(jcb);
  setLayout(new FlowLayout());
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(400, 400);
  setVisible(true);
 }
 public static void main(String[] args)
 {
  new Test();
 }
}
```



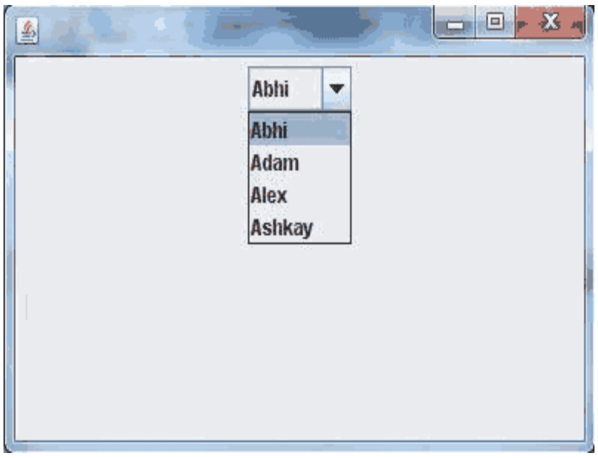
JComboBox:

Combo box is a combination of text fields and drop-down list. **JComboBox** component is used to create a combo box in Swing.

Example using JComboBox

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Test extends JFrame
{
 String name[] = {"Abhi","Adam","Alex","Ashkay"};  //list of name.
 public Test()
 {
```

```
 JComboBox jc = new JComboBox(name);   //initialzing combo box with list of name.
 add(jc); //adding JComboBox to frame.
 setLayout(new FlowLayout());
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 setSize(400, 400);
 setVisible(true);
}
public static void main(String[] args)
{
 new Test();
}
}
```



Layout Managers:

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

---

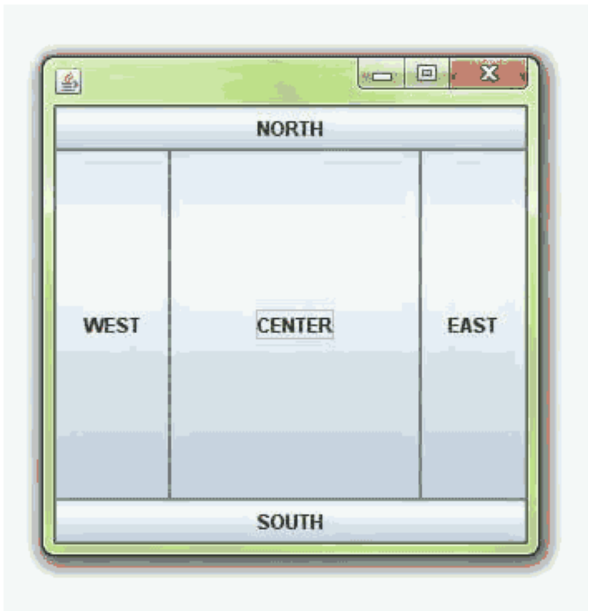
1. Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window.

Ex:

```
import java.awt.*;
import javax.swing.*;
public class Border {
JFrame f;
Border(){
   f=new JFrame();
   JButton b1=new JButton("NORTH");;
   JButton b2=new JButton("SOUTH");;
   JButton b3=new JButton("EAST");;
   JButton b4=new JButton("WEST");;
   JButton b5=new JButton("CENTER");;
   f.add(b1,BorderLayout.NORTH);
   f.add(b2,BorderLayout.SOUTH);
   f.add(b3,BorderLayout.EAST);
   f.add(b4,BorderLayout.WEST);
   f.add(b5,BorderLayout.CENTER);
```

```
    f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new Border();
}
}
```

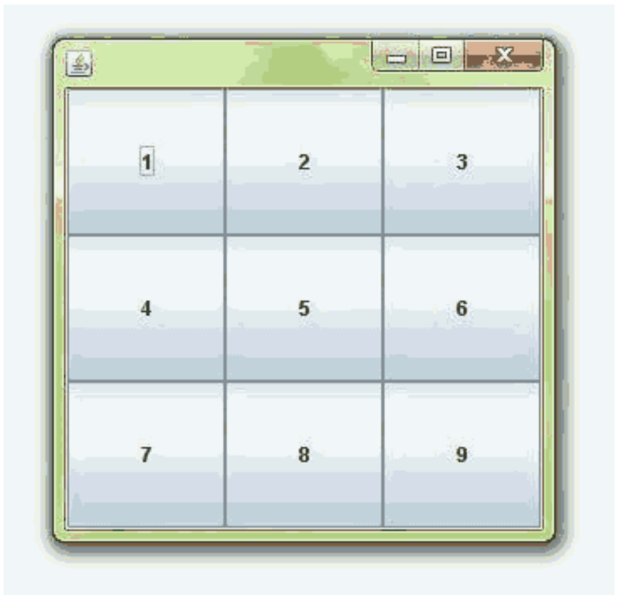


2.Java GridLayout

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Ex:

```
import java.awt.*;
import javax.swing.*;
public class MyGridLayout{
JFrame f;
MyGridLayout(){
    f=new JFrame();
    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");
    JButton b6=new JButton("6");
    JButton b7=new JButton("7");
    JButton b8=new JButton("8");
    JButton b9=new JButton("9");
    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.add(b6);f.add(b7);f.add(b8);f.add(b9);
     f.setLayout(new GridLayout(3,3));
    //setting grid layout of 3 rows and 3 columns
     f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new MyGridLayout();
}
}
```



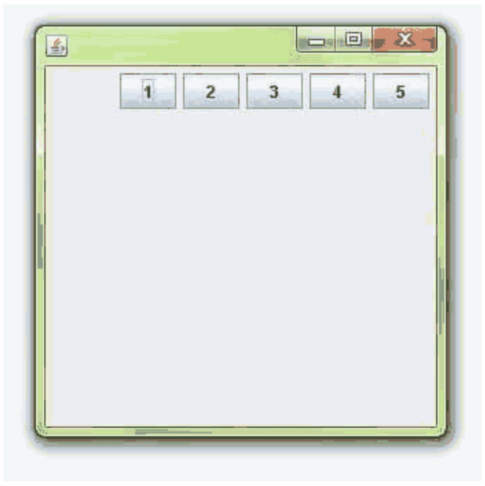
3. Java FlowLayout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

Ex:

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout{
```

```
JFrame f;
MyFlowLayout(){
    f=new JFrame();
     JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");
   f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
   f.setLayout(new FlowLayout(FlowLayout.RIGHT));
    //setting flow layout of right alignment
   f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new MyFlowLayout();
}
```



}

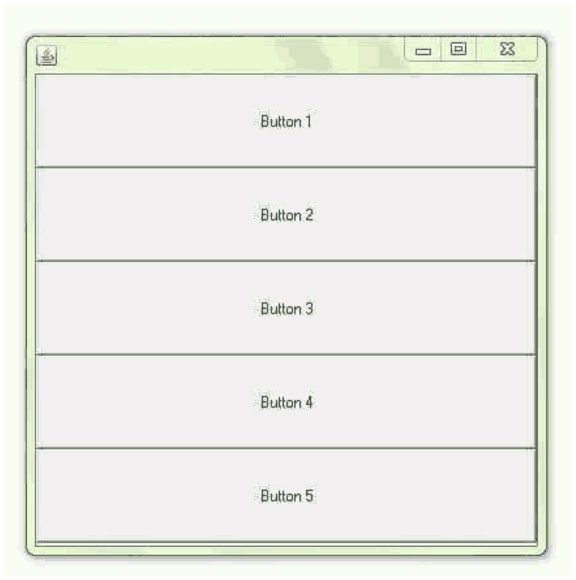
4.Java BoxLayout

The BoxLayout is used to arrange the components either vertically or horizontally.

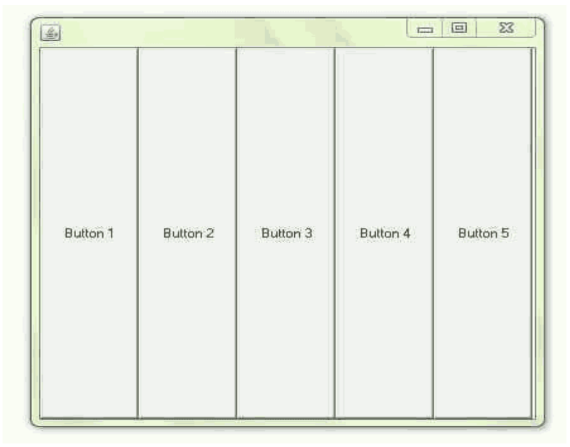
Ex:

```
import java.awt.*;
import javax.swing.*;
 public class BoxLayoutExample1 extends Frame {
```

```
 Button buttons[];
 public BoxLayoutExample1 () {
   buttons = new Button [5];
   for (int i = 0;i<5;i++) {
   buttons[i] = new Button ("Button " + (i + 1));
   add (buttons[i]);
    }
setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
setSize(400,400);
setVisible(true);
}
public static void main(String args[]){
BoxLayoutExample1 b=new BoxLayoutExample1();
}
}
```



Example of BoxLayout class with X-AXIS

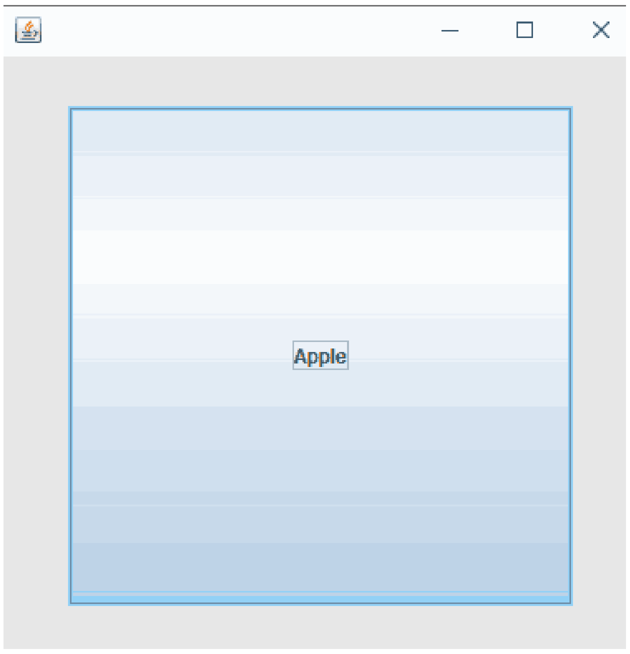


5. Java CardLayout

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout

Ex:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
CardLayoutExample(){
c=getContentPane();
card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
c.setLayout(card);
    b1=new JButton("Apple");
     b2=new JButton("Boy");
     b3=new JButton("Cat");
     b1.addActionListener(this);
     b2.addActionListener(this);
     b3.addActionListener(this);
    c.add("a",b1);c.add("b",b2);c.add("c",b3);
  }
  public void actionPerformed(ActionEvent e) {
   card.next(c);
   }
   public static void main(String[] args) {
       CardLayoutExample cl=new CardLayoutExample();
       cl.setSize(400,400);
       cl.setVisible(true);
       cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```



6. Java GridBagLayout

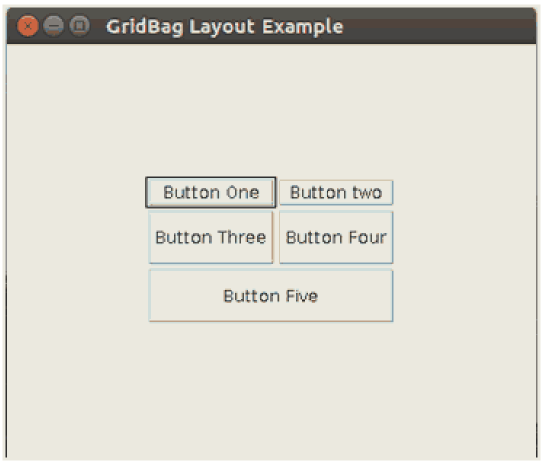
The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.

The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

Ex:

```
import java.awt.Button;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.*;
public class GridBagLayoutExample extends JFrame{
  public static void main(String[] args) {
      GridBagLayoutExample a = new GridBagLayoutExample();
    }
    public GridBagLayoutExample() {
    GridBagLayoutgrid = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(grid);
      setTitle("GridBag Layout Example");
      GridBagLayout layout = new GridBagLayout();
      this.setLayout(layout);
       gbc.fill = GridBagConstraints.HORIZONTAL;
       gbc.gridx = 0;
      gbc.gridy = 0;
      this.add(new Button("Button One"), gbc);
  gbc.gridx = 1;
  gbc.gridy = 0;
  this.add(new Button("Button two"), gbc);
  gbc.fill = GridBagConstraints.HORIZONTAL;
  gbc.ipady = 20;
  gbc.gridx = 0;
  gbc.gridy = 1;
  this.add(new Button("Button Three"), gbc);
  gbc.gridx = 1;
  gbc.gridy = 1;
  this.add(new Button("Button Four"), gbc);
  gbc.gridx = 0;
  gbc.gridy = 2;
  gbc.fill = GridBagConstraints.HORIZONTAL;
  gbc.gridwidth = 2;
  this.add(new Button("Button Five"), gbc);
      setSize(300, 300);
```

```
        setPreferredSize(getSize());
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```



7. GroupLayout

As the name implies, GroupLayout manages the layout of groups and places them in different positions. It consists of two type of groups: sequential and parallelgroup.

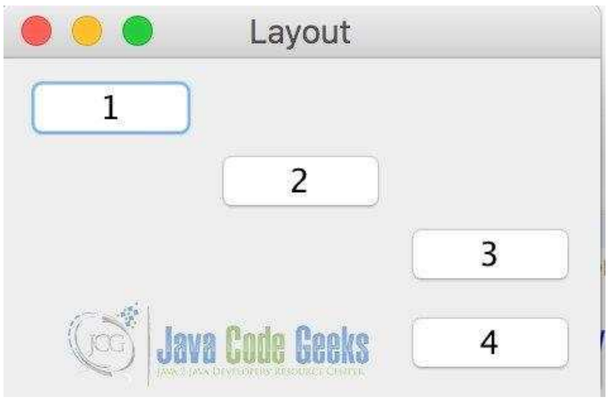
For sequential group arrangement, the components are placed very similar to BoxLayout or FlowLayout, one after another. The position of each component is according to the order of the component.

For parallel group arrangement, components are placed on top of each other at the same place. They can be baseline-, top- or bottom-aligned at vertical, or left-, right-, center-aligned at horizontal axis.

In the following example, four buttons are created with button 1, 2, 3 are following the sequential pattern, while button 3, 4 are grouped together.

GroupLayoutExample.java

```
package javaCodeGeeks;
import javax.swing.GroupLayout;
/*
 * A Java swing GroupLayout example
 */
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class GroupLayoutExample {
    public static void main(String[] args) {
        // Create and set up a frame window
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("Layout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Define new buttons with different width on help of the ---
        JButton jb1 = new JButton("1");
        JButton jb2 = new JButton("2");
        JButton jb3 = new JButton("3");
        JButton jb4 = new JButton("4");
        // Define the panel to hold the buttons
        JPanel panel = new JPanel();
        panel.setSize(300, 300);
        GroupLayout layout = new GroupLayout(panel);
        layout.setAutoCreateGaps(true);
        layout.setAutoCreateContainerGaps(true);
        panel.setLayout(layout);
        // Set for horizontal and vertical group
        layout.setHorizontalGroup(layout.createSequentialGroup().addComponent(jb1).addCom
ponent(jb2)
                .addGroup(layout.createSequentialGroup().addGroup(layout
                .createParallelGroup(GroupLayout.Alignment.LEADING).addComponent(jb3).add
Component(jb4))));
        layout.setVerticalGroup(layout.createSequentialGroup().addComponent(jb1).addCompo
nent(jb2).addComponent(jb3).addComponent(jb4));
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```



8. Java SpringLayout

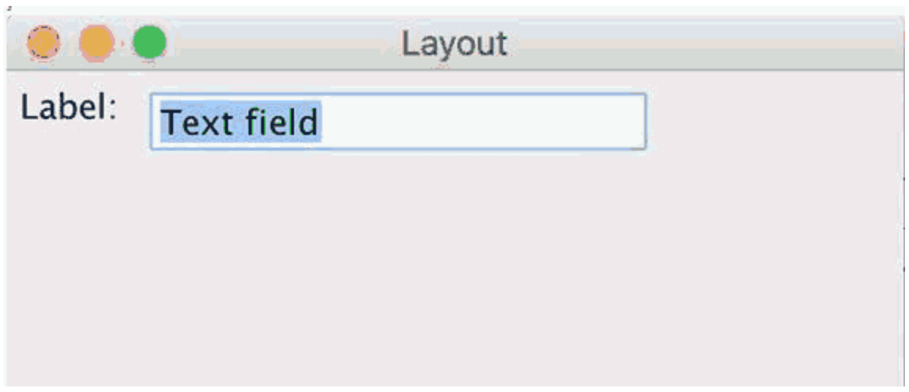
Similar to the name,Spring Layout manages the layout of its children/Spring. Every child of Spring object controls the vertical or horizontal distance between the two components edges. In addition, for every child, it has exactly one set of constraint associated with it.

In the example below, we have constructed a label with a textfield and put constraints on the edge of the two components.

Ex:

```
package javaCodeGeeks;
```

```
 /*
 * A Java swing SpringLayout example
 */
 import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SpringLayout;
 public class SpringLayoutExample {
     public static void main(String[] args) {
        // Create and set up a frame window
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("Layout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         // Define the panel to hold the components
        JPanel panel = new JPanel();
        SpringLayout layout = new SpringLayout();
        JLabel label = new JLabel("Label: ");
        JTextField text = new JTextField("Text field", 15);
        panel.setSize(300, 300);
        panel.setLayout(layout);
        panel.add(label);
        panel.add(text);
       // Put constraint on components
        layout.putConstraint(SpringLayout.WEST, label, 5, SpringLayout.WEST, panel);
        layout.putConstraint(SpringLayout.NORTH, label, 5, SpringLayout.NORTH, panel);
        layout.putConstraint(SpringLayout.WEST, text, 5, SpringLayout.EAST, label);
        layout.putConstraint(SpringLayout.NORTH, text, 5, SpringLayout.NORTH, panel);
       // Set the window to be visible as the default to be false
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
 }
```



9.ScrollPaneLayout:

The layout manager used by JScrollPane. JScrollPaneLayout is responsible for nine components: a viewport, two scrollbars, a row header, a column header, and four "corner" components.

Ex:

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
public class ScrollPaneDemo extends JFrame
{
public ScrollPaneDemo() {
super("ScrollPane Demo");
ImageIcon img = new ImageIcon("child.png");
JScrollPane png = new JScrollPane(new JLabel(img));
getContentPane().add(png);
setSize(300,250);
setVisible(true);
}
public static void main(String[] args) {
new ScrollPaneDemo();
}
}
```

J Scroll Pane: Just now discussed ….

Split Pane:

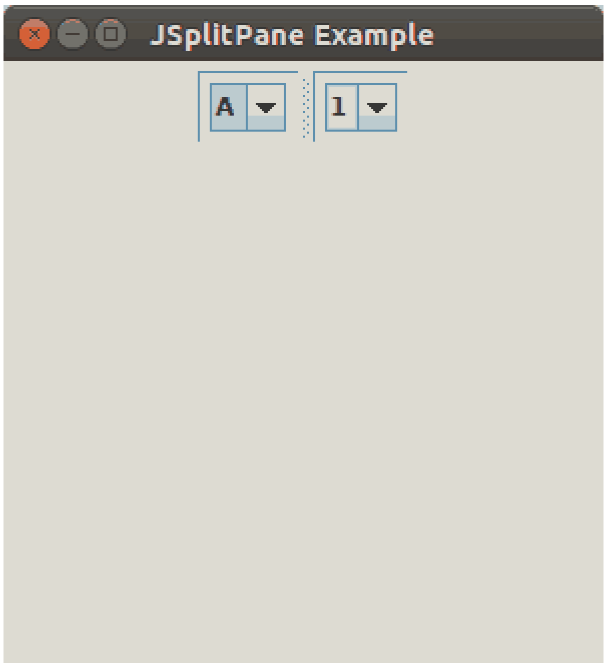
JSplitPane is used to divide two components. The two components are divided based on the look and feel implementation, and they can be resized by the user.

The two components in a split pane can be aligned left to right using JSplitPane.HORIZONTAL_SPLIT, or top to bottom using JSplitPane.VERTICAL_SPLIT. When the user is resizing the components the minimum size of the components is used to determine the maximum/minimum position the components can be set to.

Ex:

```
import java.awt.FlowLayout;
import java.awt.Panel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JSplitPane;
public class JSplitPaneExample {
   private static void createAndShow() {
      // Create and set up the window.
      final JFrame frame = new JFrame("JSplitPane Example");
      // Display the window.
      frame.setSize(300, 300);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // set flow layout for the frame
```

```
        frame.getContentPane().setLayout(new FlowLayout());
        String[] option1 = { "A","B","C","D","E" };
        JComboBox box1 = new JComboBox(option1);
        String[] option2 = {"1","2","3","4","5"};
        JComboBox box2 = new JComboBox(option2);
        Panel panel1 = new Panel();
        panel1.add(box1);
        Panel panel2 = new Panel();
        panel2.add(box2);
        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, panel1, pane
l2);
        // JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
        // panel1, panel2);
        frame.getContentPane().add(splitPane);
    }
    public static void main(String[] args) {
        // Schedule a job for the event-dispatching thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShow();
            }
        });
    }
}
```



JTabbed Pane:

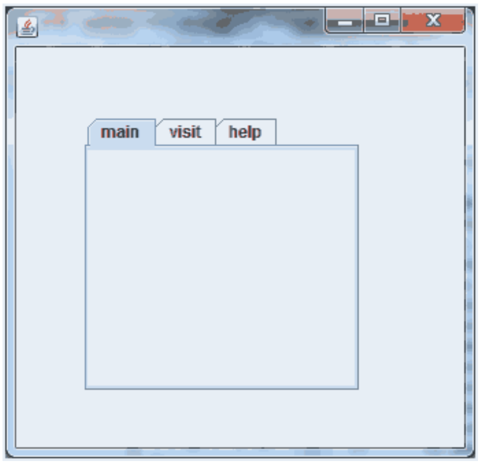
The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.

Sy:

public class JTabbedPane **extends** JComponent **implements** Serializable, Accessible, Swing Constants

Ex:

```
import javax.swing.*;
public class TabbedPaneExample {
JFrame f;
TabbedPaneExample(){
    f=new JFrame();
    JTextArea ta=new JTextArea(200,200);
    JPanel p1=new JPanel();
    p1.add(ta);
    JPanel p2=new JPanel();
    JPanel p3=new JPanel();
    JTabbedPane tp=new JTabbedPane();
    tp.setBounds(50,50,200,200);
    tp.add("main",p1);
    tp.add("visit",p2);
    tp.add("help",p3);
    f.add(tp);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
public static void main(String[] args) {
    new TabbedPaneExample();
}}
```


main
visit
help