UNIT_II

JAVA

❖ OOPs (Object-Oriented Programming System):

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

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

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

1. Object:

- An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.
- 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.

2. Class:

- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.
- Collection of objects is called class. It is a logical entity.

3. Inheritance:

- Inheritance is a acquiring properties from one class(base class) to another class(derived class) is also known as inheritance.
- It provides code reusability. It is used to achieve runtime polymorphism.

4. Polymorphism:

- If one task is performed in different ways or many forms, it is known as polymorphism.
- For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

5.Abstraction:

- Hiding internal details and showing functionality is known as abstraction.
- In Java, we use abstract class and interface to achieve abstraction.

7. Encapsulation:

- Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

❖ **Features or Buzzwords of Java:**

1) Simple:

- Java's syntax is simple and easy to learn, especially for those familiar with C or C++.
- It eliminates complex features like pointers and multiple inheritances, making it easier to write, debug, and maintain code**.**

2) Object Oriented:

- Java is an object-oriented language**,** promoting the use of objects and **c**lasses**.**

- Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

The Six concepts of Object-Oriented programming are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

3) Robust:

- Java language is robust which means reliable.
- It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language.

4) Platform Independent:

- Compiler converts source code to byte code and then the JVM executes the byte code generated by the compiler.
- This byte code can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa

5) Secure:

- Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

6) Multi Threading:

- Java multithreading feature makes it possible to write program that can do many tasks simultaneously.
- Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

7) Portable:

- Java code written on one machine can be run on another machine.
- The platform-independent feature of java in which its platform-independent byte code can be taken to any platform for execution makes java portable.

8) High Performance:

- Java is an interpreted language, so it will never be as fast as a compiled language like C or C++.
- But, Java enables high performance with the use of just-in-time compiler.

9) Distributed:

- Java is also a distributed language. Programs can be designed to run on computer networks.
- Java has a special class library for communicating using TCP/IP protocols.
- Creating network connections is very much easy in Java as compared to C/C++.

10) Architectural Neutral:

- Compiler generates byte codes, which have nothing to do with a particular computer architecture, hence a Java program is easy to interpret on any machine.

11) Interpreted:

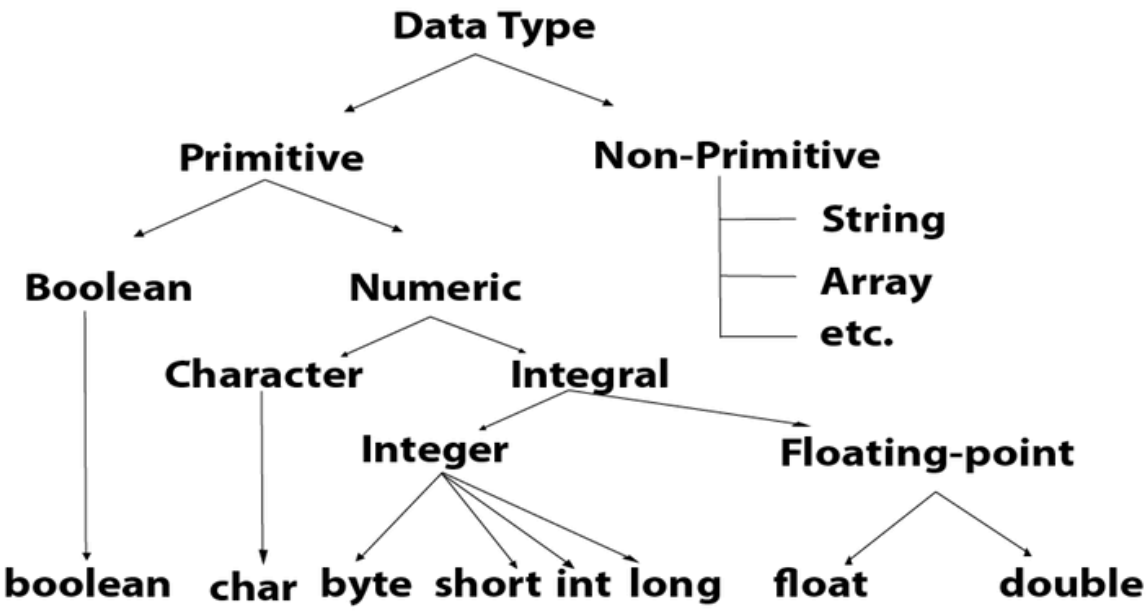
- In terms of execution, Java programs undergo both compilation and interpretation processes.
- The source code is first compiled into byte code by the Java compiler.
- Then this byte code is interpreted by the JVM when the program runs.

12)Dynamic:

- Java is a dynamic language. It supports the dynamic loading of classes.
- It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

❖ **Data Types:**

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:



1. **Primitive data types:**
    - The primitive data types are the building blocks of data manipulation.
    - The primitive data types include Boolean, char, byte, short, int, long, float and double.

Boolean Data Type:

- The Boolean data type is used to store only two possible values: true and false.
- The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

EXAMPLE: Boolean one = false

Char Data Type:

- The char data type is a single 16-bit Unicode character.
- The char data type is used to store characters.

Example: char letter A = 'A'

Byte Data Type:

- The byte data type is an 8-bit signed two's complement integer.
- Its value-range lies between -128 to 127

Example: byte a = 10, byte b = -20

Short Data Type:

- The short data type is a 16-bit signed two's complement integer.
- Its value-range lies between -32,768 to 32,767.

Example: short s = 10000, short r = -5000

Int Data Type:

- The int data type is a 32-bit signed two's complement integer.
- Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) .

Example: int a = 100000, int b = -200000

PROGRAM:

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

```
        {
        char a = 'G';
         int i = 89;
        byte b = 4;
        short s = 56;
      double d = 4.355453532;
      float f = 4.7333434f;
      long l = 12121;
      System.out.println("char: " + a);
      System.out.println("integer: " + i);
      System.out.println("byte: " + b);
      System.out.println("short: " + s);
      System.out.println("float: " + f);
      System.out.println("double: " + d);
      System.out.println("long: " + l);
   }
}
```

OUTPUT:

```
      char: G
   integer: 89
   byte: 4
   short: 56
   float: 4.7333436
   double: 4.355453532
   long: 12121
```

2. Non-Primitive data types:

- Unlike primitive data types, these are not predefined. These are user-defined data types created by programmers.
- These data types are used to store multiple values. They are strings, objects, arrays, etc.

There are five types of non-primitive data types in Java. They are as follows:

1. Class
2. Object
3. String
4. Array
5. Interface

1. Class & object:

Class:

- A class is a user defined data type i.e. it is created by the user.
- It acts a template to the data which consists of member variables and methods.

object:

- An object is the variable of the class, which can access the elements of class i.e. methods and variables.

Example:

```
public class ClassExample {
int a = 20;
int b = 10;
int c;
public void add () {
int c = a + b;
```

```
System.out.println("Addition of numbers is: " + c);
}
public void sub () {
int c = a - b;
System.out.println("Subtraction of numbers is: " + c);
}
public static void main (String[] args) {
ClassExample obj = new ClassExample();
obj.add();
obj.sub();
}
}
```

Output:

Addition of numbers is: 30

Subtraction of numbers is: 10

2. Interface:

- An interface is similar to a class however the only difference is that its methods are abstract by default i.e. they do not have body.
- An interface has only the final variables and method declarations. It is also called a fully abstract class.

Example:

```
interface CalcInterface {
void multiply();
void divide();
}
public class InterfaceExample implements CalcInterface {
int a = 10;
```

```
int b = 20;
int c;
public void multiply() {
int c = a * b;
System.out.println("Multiplication of numbers is: " + c);
}
public void divide() {
int c = a / b;
System.out.println("Division of numbers is: " + c);
}
public static void main (String[] args)  {
InterfaceExample obj = new InterfaceExample();
obj.multiply();
obj.divide();
}
}
```

Output:

```
Multiplication of numbers is: 200
Division of numbers is: 0
```

3. String:

- A string represents a sequence of characters for example "Javatpoint", "Hello world", etc.

Example:

```
public class StringExample {
public static void main(String[] args) {
String str = "Hello! This is example of String type";
String subStr = str.substring(0,14);
```

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

Output:

Hello! This is

4. Array:

- An array is a data type which can store multiple homogenous variables i.e., variables of same type in a sequence.
- They are stored in an indexed manner starting with index 0.

Example:

```
import java.io. * ;
import java.util. * ;
public class ArrayExample {
public static void main(String[] args) {
int i;
Scanner sc = new Scanner(System. in );
int arr[] = {1, 2, 3, 6, 9};
int arr1[] = new int[5];
System.out.println("Enter the numbers (size = 5) :");
for (i = 0; i < 5; i++) {
arr1[i] = sc.nextInt();
}
System.out.println("Previous array with initialized size is: ");
for (i = 0; i < 5; i++) {
System.out.print(arr[i] + " ");
}
System.out.println("\nThe new array we have entered is:");
```

```
    for (i = 0; i < 5; i++) {
    System.out.print(arr1[i] + " ");
    }
    }
   }
```

Output:

Enter the numbers (size = 5) :

56

43

22

1

9

Previous array with initialized size is:

1 2 3 6 9

The new array we have entered is:

56 43 22 1 9

- ❖ **Variables and arrays:**

- A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.Variable is a name of memory location.
- A variable is the name of a reserved area allocated in memory. It is a combination of "vary + able" which means its value can be changed.

  int data=50;//Here data is variable

Types of Variables

There are three types of variables ,

1. local variable
2. instance variable

3. static variable

1) Local Variable:

- A variable declared inside the body of the method is called local variable.
- You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable:

- A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.
- It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable:

- A variable that is declared as static is called a static variable.
- It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class.
- Memory allocation for static variables happens only once when the class is loaded in the memory.

Syntax:

```
public class A
{
static int m=100;//static variable
void method()
{
int n=90;//local variable
}
public static void main(String args[])
```

```
{
int data=50;//instance variable
}
}//end of class
```

Example :

```
public class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}
}
```

Output: 20

Array :

- Array is a collection of similar data types or homogeneous elements. This means that all the elements in the array are of the same data type.
- **Example:** This is an array of seven elements. All the elements are integers and homogeneous. In this case, as there are seven elements, the index is from zero to six.

  There are three main features of an array:

1. **Dynamic allocation:** In arrays, the memory is created dynamically, which reduces the amount of storage required for the code.
2. **Elements stored under a single name:** All the elements are stored under one name. This name is used any time we use an array.

3. **Occupies contiguous location:** The elements in the arrays are stored at adjacent positions. This makes it easy for the user to find the locations of its elements.

Advantages of Arrays:

- Java arrays enable you to access any element randomly with the help of indexes
- It is easy to store and manipulate large data sets

Disadvantages of Arrays:

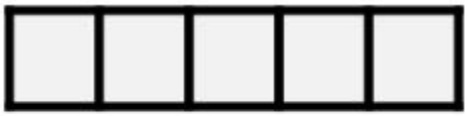
- The size of the array cannot be increased or decreased once it is declared—arrays have a fixed size
- Java cannot store heterogeneous data. It can only store a single type of primitives

Types of Arrays

There are three types of arrays. We use these types of arrays as per the requirement of the program. These are:

1. One-dimensional Array:

Also known as a linear array, the elements are stored in a single row. For example: In this example, we have an array of five elements. They are stored in a single line or adjacent memory locations.



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

Example:

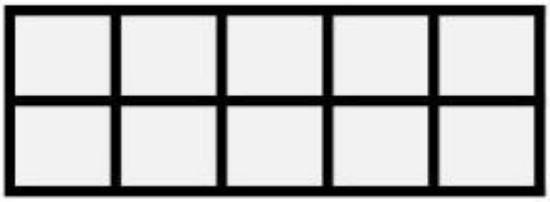
```
import java.io.*;
class GFG {
public static void main (String[] args) {
```

```
int [] arr=new int [4];
System.out.println("Array Size:"+arr.length);
}
}
```

Output: Array Size:4

2. Two-dimensional Array:

Two-dimensional arrays store the data in rows and columns:



```
int[][] Array = new int[2][5];
```

In this, the array has two rows and five columns. The index starts from 0,0 in the left-upper corner to 1,4 in the right lower corner.

Example:

```
public class GFG {
public static void main(String[] args) {
int[] arr = new int[2];
arr[0] = 10;
arr[1] = 20;
for (int i = 0; i < arr.length; i++)
System.out.println(arr[i]);
}
}
```

Output:

10

20

3. Multi-dimensional Array:

This is a combination of two or more arrays or nested arrays. We can even use more than two rows and columns .

Example:

```
import java.io.*;
class GFG {
public static void main(String[] args) {
nt[][] arr = new int[3][3];
System.out.println("Number of Rows:"+ arr.length);
System.out.println("Number of Columns:"+arr[0].length);
}
}
```

Output:

Number of Rows:3
Number of columns:3

- **Operators:**

Operators in Java are the symbols used for performing specific operations. There are different types of operators,

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational /comparison Operators
5. Logical Operators
6. Ternary Operator

7. Bitwise Operators
8. Shift Operators

1. Arithmetic Operators

- Arithmetic operators are used to perform addition, subtraction, multiplication, and division.

They act as basic mathematical operations.

Example:

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);
 System.out.println(a-b);
System.out.println(a*b);
System.out.println(a/b);
System.out.println(a%b);
}}
```

Output:

15

5

50

2

0

2. Unary Operator:

- The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:
- **Unary minus**, used for negating the values.

- **+ : Unary plus** indicates the positive value (numbers are positive without this, however).
- **++ : Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.
    - **Post-Increment:** Value is first used for computing the result and then incremented.
    - **Pre-Increment:** Value is incremented first, and then the result is computed.
- **– – : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.
        - **Post-decrement:** Value is first used for computing the result and then decremented.
        - **Pre-Decrement: The value** is decremented first, and then the result is computed.

Example:

```
import java.io.*;
class Unary{
public static void main(String[] args)
{
int a = 10;
int b = 20;
System.out.println("Postincrement : " + (a++));
System.out.println("Preincrement : " + (++a));
System.out.println("Postdecrement : " + (b--));
System.out.println("Predecrement : " + (--b));
}
```

}

Output:

Postincrement : 10

Preincrement : 12

Postdecrement : 20

Predecrement : 18

3. Assignment Operator:

- Assignment operator is used to assign a value to any variable. It is used to assign the value on its right to the operand on its left.

- **+=:** for adding the left operand with the right operand and then assigning it to the variable on the left.
- -=: for subtracting the right operand from the left operand and then assigning it to the variable on the left.
- *= :for multiplying the left operand with the right operand and then assigning it to the variable on the left.
- /=: for dividing the left operand by the right operand and then assigning it to the variable on the left.
- **%=:** for assigning the modulo of the left operand by the right operand and then assigning it to the variable on the left.

Example:

```
public class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
```

```
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

```
13
9
18
9
```

4.Relational /comparison Operators:

- comparison operators are used to compare two values (or variables)and return value of a comparison is either true or false.

Some of the relational operators are-

- **==, Equal to** returns true if the left-hand side is equal to the right-hand side.
- **!=, Not Equal to** returns true if the left-hand side is not equal to the right-hand side.
- **<, less than:** returns true if the left-hand side is less than the right-hand side.
- **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.
- **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
- **>=, Greater than or equal to** returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

```
import java.io.*;
class GFG {
```

```
public static void main(String[] args) {
int a = 10;
int b = 3;
int c = 5;
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("a == c: " + (a == c));
System.out.println("a != c: " + (a != c));
}
}
```

Output:

a > b: true

a < b: false

a >= b: true

a <= b: false

a == c: false

a != c: true

5. Logical Operators:

- These operators are used to perform “logical AND” and “logical OR” operations, Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

Example:

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

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

Output:

```
x && y: false
x || y: true
!x: false
```

6. Ternary operator:

- The ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary.

Example:

```
public class operators {
public static void main(String[] args){
int a = 20, b = 10, c = 30, result;
result= ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
System.out.println("Max of three numbers = "+ result);
}
}
```

Output:

```
Max of three numbers = 30
```

7. Bitwise Operators:

- These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types.
- They are used when performing update and query operations of the Binary indexed trees.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit-by-bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's complement representation of the input value, i.e., with all bits inverted.

Example:

```
import java.io.*;
class GFG {
public static void main(String[] args)
{
int d = 0b1010;
int e = 0b1100;
System.out.println("d & e: " + (d & e));
System.out.println("d | e: " + (d | e));
System.out.println("d ^ e: " + (d ^ e));
System.out.println("~d: " + (~d));
System.out.println("d << 2: " + (d << 2));
System.out.println("e >> 1: " + (e >> 1));
System.out.println("e >>> 1: " + (e >>> 1));
}
}
```

Output:

```
d & e: 8
d | e: 14
d ^ e: 6
```

~d: -11
d << 2: 40
e >> 1: 6
e > 1: 6

8. Shift Operators:

- These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively.
- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.
- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of the initial number. Similar effect to dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

  Example:

```
import java.io.*;
class GFG {
public static void main(String[] args){
int a = 10;
System.out.println("a<<1 : " + (a << 1));
System.out.println("a>>1 : " + (a >> 1));
}
}
```

  Output:

```
a<<1 : 20
```

a>>1 : 5

❖ Control Statements:

- Control statements to control the flow of execution of a program based on certain conditions.
- These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

Java provides three types of control flow statements,

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - for loop
    - for-each loop
    - while loop
    - do while loop
3. Jump statements
    - break statement
    - continue statement

1. Decision-Making statements:

- Decision making in programming is similar to decision-making in real life.
- In programming also face some situations where we want a certain block of code to be executed when some condition is fulfilled.

There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

- The ifstatement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition.
- The condition of the If statement gives a Boolean value, either true or false.

There are four types of if-statements,

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

1) Simple if statement:

- It is the most basic statement among all control flow statements.
- . It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statements is executed otherwise not.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
```

Example:

```
importjava.util.*;
classIfDemo {
public static void main(String args[]){
int i = 10;
if (i < 15)
System.out.println("10 is less than 15");
}
```

```
}
```

Output:

```
10 is less than 15
```

2) if-else Statement:

- The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block.
- The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Example:

```
importjava.util.*;
classIfElseDemo {
public static void main(String args[])
{
int i = 10;
if (i < 15)
System.out.println("i is smaller than 15");
else
System.out.println("i is greater than 15");
}
}
```

Output:

i is smaller than 15

3) if-else-if ladder:

- The if-else-if statement contains the if-statement followed by multiple else-if statements.
- The if statements are executed from the top down.

Syntax :

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Example.

```
importjava.util.*;
classifelseifDemo {
public static void main(String args[]){
int i = 20;
if (i == 10)
System.out.println("i is 10");
else if (i == 15)
System.out.println("i is 15");
else if (i == 20)
System.out.println("i is 20");
```

```
else
System.out.println("i is not present");
}
}
```

Output:

```
i is 20
```

4. Nested if-statement:

- In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax :

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

Example:

```
importjava.util.*;
classNestedIfDemo {
public static void main(String args[]){
int i = 10;
if (i == 10 || i<15) {
if (i < 15)
```

```
System.out.println("i is smaller than 15");
if (i < 12)
System.out.println(
"i is smaller than 12 too");
} else{
System.out.println("i is greater than 15");
}
}
}
```

Output:

i is smaller than 15

i is smaller than 12 too

2.Switch Statement:

- Switch statements are similar to if-else-if statements.
- The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched.

Syntax:

```
switch (expression){
case value1:
statement1;
break;
.
.
.
casevalueN:
statementN;
break;
```

```
default:
default statement;
}
```

Example:

```
import java.io.*;
class GFG {
public static void main (String[] args) {
intnum=20;
switch(num){
case 5 : System.out.println("It is 5");
break;
case 10 : System.out.println("It is 10");
break;
case 15 : System.out.println("It is 15");
break;
case 20 : System.out.println("It is 20");
break;
default: System.out.println("Not present");
}
}
}
```

Output:

```
It is 20
```

Loop Statements:

- Loop statements are used to execute the set of instructions in a repeated order.

- The execution of the set of instructions depends upon a particular condition.

These arefour types of loops that execute similarly.

1. for loop
2. for-each loop
3. while loop
4. do-while loop

1. for loop:

- The Java for loop repeats the execution of a set of Java statements.
- A for loop executes a block of code as long as some condition is true.

Syntax:

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

Example:

```
import java.io.*;
class GFG {
public static void main (String[] args) {
for (int i=0;i<=10;i++){
System.out.println(i);
}
}
}
```

Output:

```
0
1
2
```

3

4

5

6

7

8

9

10

2. for-each loop:

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

syntax:

```
public class Calculation {
public static void main(String[] args) {
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

Java

C

C++

Python

3.while loop:

- The while statement or loop continually executes a block of statements while a particular condition is true.
- The while statement continues testing the expression and executing its block until the expression evaluates to false.

Syntax :

```
while(condition){
//looping statements
}
```

Example:

```
public class Calculation {
public static void main(String[] args) {
int x = 1;
while (x <= 4) {
System.out.println("Value of x:" + x);
x++;
}
}
}
```

Output:

```
Value of x: 1
Value of x: 2
Value of x: 3
Value of x: 4
```

4.do-while loop:

- The do-while loop checks the condition at the end of the loop after executing the loop statements.
- When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

Syntax:

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

Example:

```
public class Calculation {
public static void main(String[] args) {
int x = 21;
do
{
System.out.println("Value of x:" + x);
x++;
}while (x < 20);
}
}
```

Output:

```
Value of x: 21
```

Jump Statements:

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program.

There are two types of jump statements in Java,

1.Break statement:

- The break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement.
- However, it breaks only the inner loop in the case of the nested loop.

Example:

```
public class BreakExample {
public static void main(String[] args) {
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

0
1
2
3
4
5
6

2. Continue statement:

- Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Example:

```
public class ContinueExample {
public static void main(String[] args) {
for(int i = 0; i<= 2; i++) {
for (int j = i; j<=5; j++) {
if(j == 4) {
continue;
}
System.out.println(j);
}
}
}
}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

❖ Inheritance:

- Inheritance is a acquiring properties from one class (base class) to another class (derived class) is also known as inheritance.
- It provides code reusability. It is used to achieve runtime polymorphism.
- Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Terms used in Inheritance:

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class.

Syntax:

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

}
```

Types of inheritance:

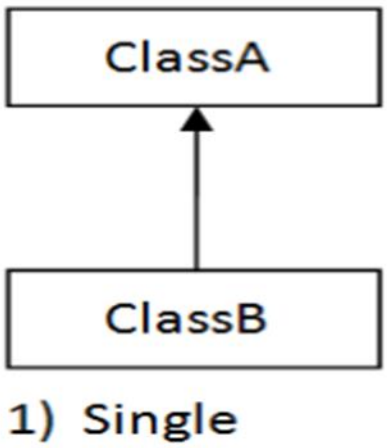
There are the different types of inheritance in java:

1. Single Inheritance

2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

1.Single Inheritance:

- In single inheritance, a sub-class is derived from only one super class.
- It inherits the properties and behavior of a single-parent class.



1) Single

Example:

```
class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){
System.out.println("barking...");}
}
classTestInheritance{
public static void main(String args[]){
```

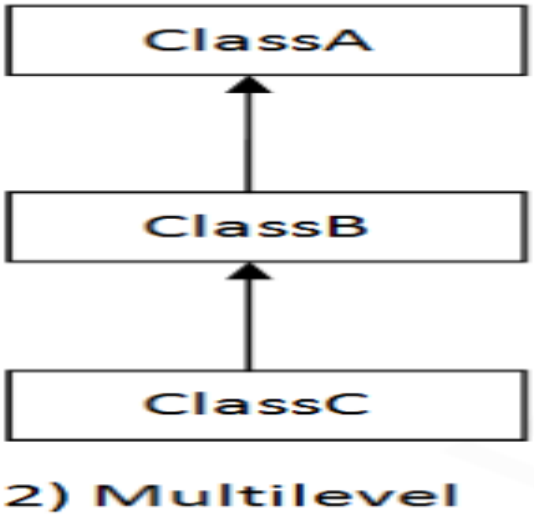
```
Dog d=new Dog();
d.bark();
d.eat();
}
```

Output:

```
barking...
eating...
```

2. Multilevel Inheritance:

- In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.
- In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



2) Multilevel

Example:

```
class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
```

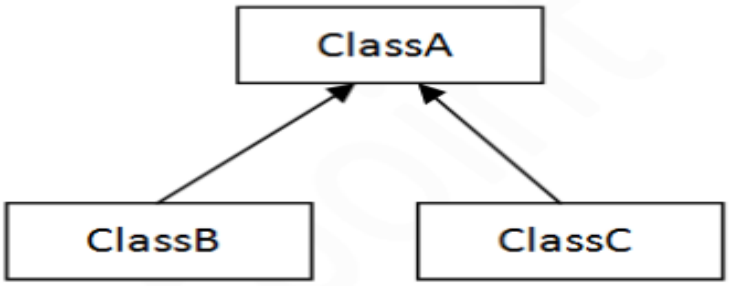
```
void bark(){
System.out.println("barking...");}
}
classBabyDog extends Dog{
void weep(){
System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

3. Hierarchical Inheritance:

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.
- In the below image, class A serves as a base class for the derived classes B, C, and D.



3) Hierarchical

Example:

```
class Animal{
void eat(){
System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){
System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){
System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
```

}}

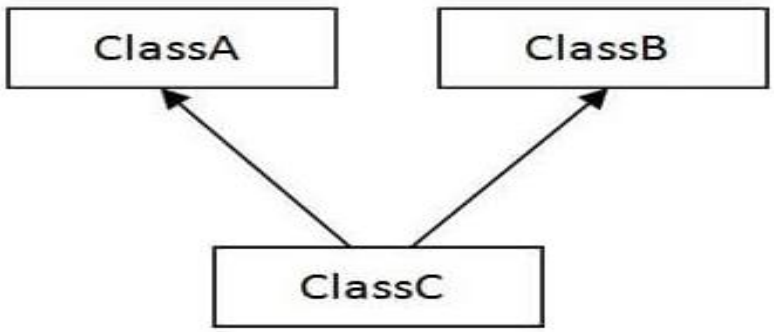
Output:

meowing...

eating...

4. Multiple Inheritance:

- In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes.
- In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



4) Multiple

Example:

```
class A{
voidmsg(){
System.out.println("Hello");}
}
class B{
voidmsg(){
System.out.println("Welcome");}
```

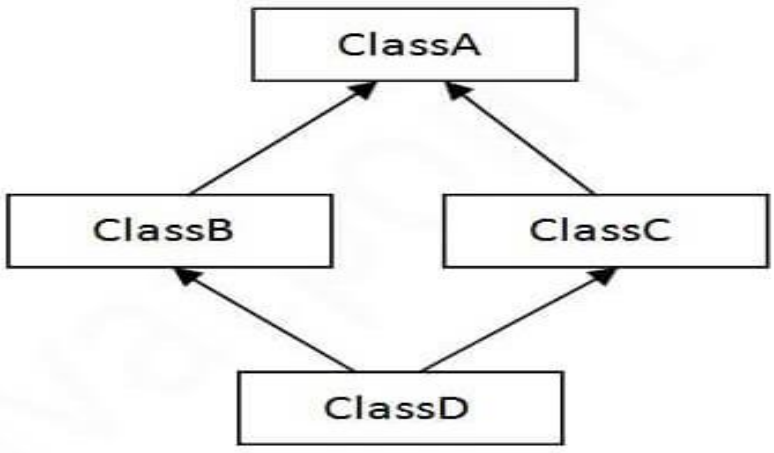
```
}
class C extends A,B{//suppose if it were
public static void main(String args[]){
  C obj=new C();
obj.msg();//Now which msg() method would be invoked?
}
}
```

Output:

Compile Time Error

5. Hybrid Inheritance:

- It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes.
- In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.



5) Hybrid

Example:

```
classSolarSystem {
```

```
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
public static void main(String args[])
{
SolarSystem s = new SolarSystem();
Earth e = new Earth();
Mars m = new Mars();
System.out.println(s instanceofSolarSystem);
System.out.println(e instanceof Earth);
System.out.println(m instanceofSolarSystem);
}
}
```

Output:

```
true
true
true
```

❖ packages and Interfaces:

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form,
    1. Built-in package
    2. User-defined package.

1. Built-in Packages:

- These packages consist of a large number of classes which are a part of Java API.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

2. User-defined package:

- These are the packages that are defined by the user.
- First we create a directory myPackage (name should be same as the name of the package).
- Then create the MyClass inside the directory with the first statement being the package names.

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

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

1) Using import packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example :

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
}
}
```

Output: Hello

2) Using packagename.classname:

- If you import package.classname then only declared class of this package will be accessible.

Example:

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;
class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
```

```
}
}
```

Output: Hello

3) Using fully qualified name:

- If you use fully qualified name then only declared class of this package will be accessible.
- Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

Example :

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
public static void main(String args[]){
pack.A obj = new pack.A();//using fully qualified name
obj.msg();
}
}
```

Output: Hello

Advantages:

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

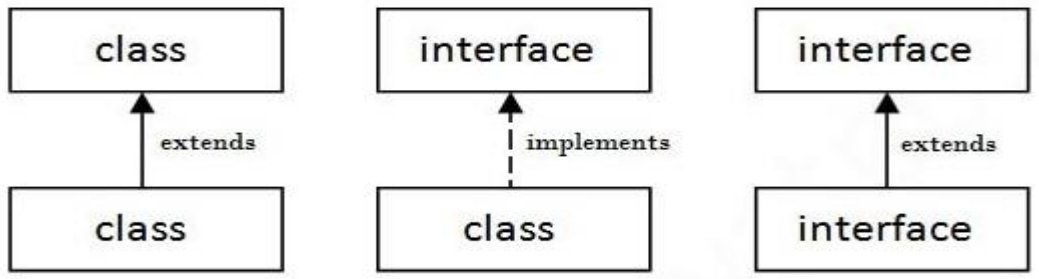
2) Java package provides access protection.
3) Java package removes naming collision.

Interfaces:

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- It cannot be instantiated just like the abstract class.
- Java Interface also represents the **IS-A relationship**.

The relationship between classes and interfaces:

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



Syntax:

```
interface <interface_name>{
// declare constant fields
// declare methods that abstract
// by default.
```

```
}
```

Example:

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){
System.out.println("Hello");}
public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

Output: Hello

Interface inheritance:

A class implements an interface, but one interface extends another interface.

Example:

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){
System.out.println("Hello");}
```

```
public void show(){
ystem.out.println("Welcome");}
public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

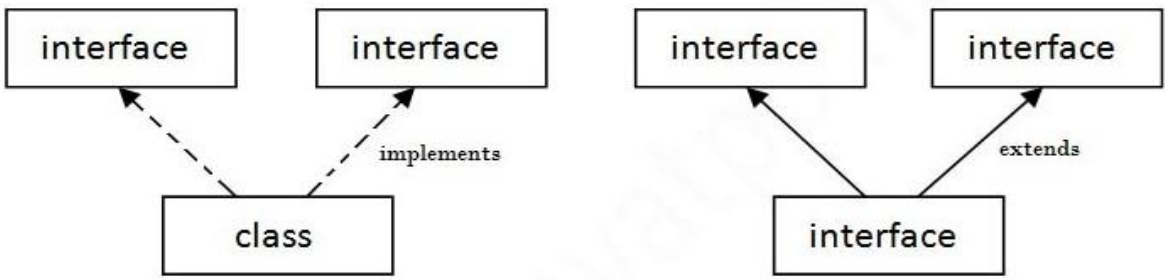
Output:

Hello

Welcome

Multiple inheritance in Java by interface:

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
interface Printable{
void print();
}
```

```
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Output:

Hello

Welcome

There are mainly three reasons to use interface. They are given below.

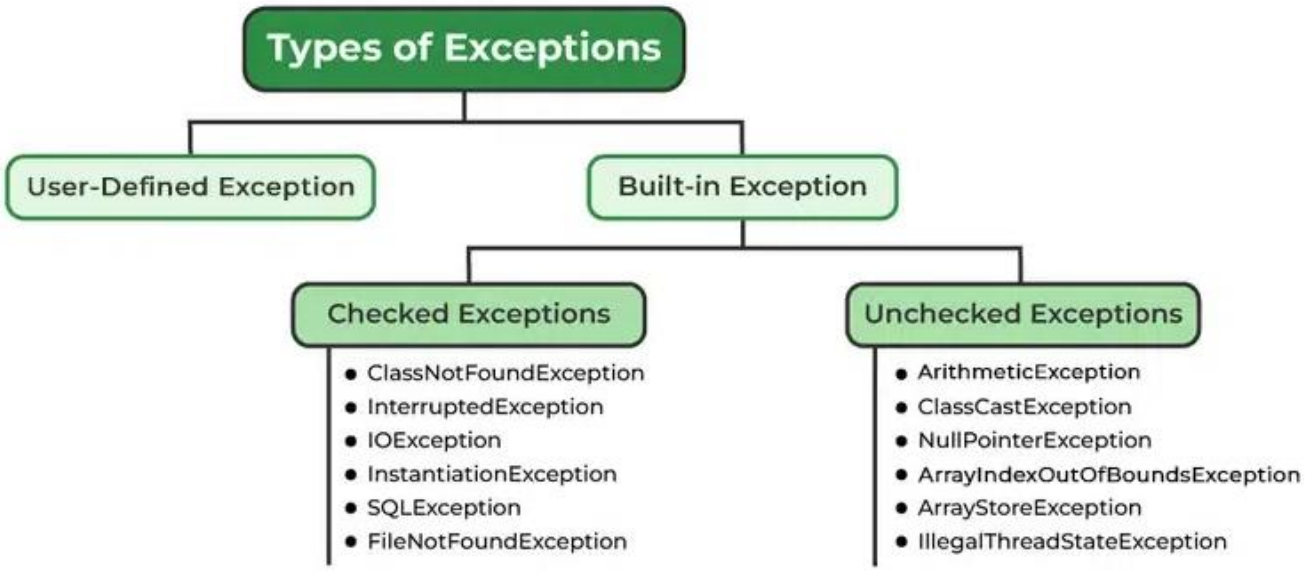
1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

❖ **Exception Handling :**

- Exception is a run time error**.** An exception is an error event that can happen during the execution of a program and disrupts its normal flow.
- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Types of Exceptions:

- Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. Built-in Exceptions
    1. Checked Exception
    2. Unchecked Exception
2. User-Defined Exceptions

1. Built-in Exceptions:

- Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

1. **Checked Exceptions:**

- Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

2. **Unchecked Exceptions**:

- unchecked exceptions are called run time exception. The compiler will not check these exceptions at compile time.

2. User-Defined Exceptions:

- Sometimes, the built-in exceptions in Java are not able to describe a certain situation.
- In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

Advantages of Exception Handling:

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types

Java Exception Handling:

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

1. try...catch block
2. finally block
3. throw and throws keyword

1. Java try...catch block:

The try-catch block is used to handle exceptions in Java.

Syntax of try...catch block:

```
try {
  // code
}
catch(Exception e) {
```

```
    // code
}
```

Example:

```
class Main {
public static void main(String[] args) {
try {
intdivideByZero = 5 / 0;
System.out.println("Rest of code in try block");
}
catch (ArithmeticException e) {
System.out.println("ArithmeticException => " + e.getMessage());
}
}
}
```

Output:

```
ArithmeticException => / by zero
```

2. Java finally block:

- The finally block is always executed no matter whether there is an exception or not.
- The finally block is optional. And, for each try block, there can be only one finally block.

Syntax:

```
try {
  //code
```

```
    }
    catch (ExceptionType1 e1) {
      // catch block
    }
    finally {
      // finally block always executes
    }
```

Example:

```
class Main {
public static void main(String[] args) {
try {
intdivideByZero = 5 / 0;
}
catch (ArithmeticException e) {
System.out.println("ArithmeticException => " + e.getMessage());
}
finally {
System.out.println("This is the finally block");
}
}
}
```

Output:

```
    ArithmeticException => / by zero
```

This is the finally block

3. Java throw and throws keyword:

- The Java throw keyword is used to explicitly throw a single exception.
- When we throw an exception, the flow of the program moves from the try block to the catch block.

Example:

```
class Main {
public static void divideByZero() {
throw new ArithmeticException("Trying to divide by 0");
  }
public static void main(String[] args) {
divideByZero();
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
atMain.divideByZero(Main.java:5)
atMain.main(Main.java:9)
```

In the above example, we are explicitly throwing the ArithmeticException using the throw keyword.

Throws keyword:

- The throws keyword is used to declare the type of exceptions that might occur within the method.
- It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;
class Main {
public static void findFile() throws IOException {
File newFile = new File("test.txt");
FileInputStream stream = new FileInputStream(newFile);
}
public static void main(String[] args) {
try {
findFile();
}
catch (IOException e) {
System.out.println(e);
}
}
}
```

Output:

java.io.FileNotFoundException: test.txt (The system cannot find the file specified)

❖ **Multithreading:**

- Multithreading in Java is a process of executing multiple threads simultaneously**.**
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.

Threads can be created by using two mechanisms:

1. Extending the Thread class

2. Implementing the Runnable Interface

1.Thread creation by extending the Thread class:

- We create a class that extends the java.lang.Thread class.This class overrides the run() method available in the Thread class.
- A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

Example:

```
classMultithreadingDemo extends Thread {
public void run(){
try {
System.out.println("Thread " +Thread.currentThread().getId()+ " is running");
}
catch (Exception e) {
System.out.println("Exception is caught");
}
}
}
public class Multithread {
public static void main(String[] args)
{
int n = 8; // Number of threads
for (int i = 0; i < n; i++) {
MultithreadingDemo object= new MultithreadingDemo();
object.start();
}
}
```

}

Output:

Thread 15 is running

Thread 14 is running

Thread 16 is running

Thread 12 is running

Thread 11 is running

Thread 13 is running

Thread 18 is running

Thread 17 is running

2.Thread creation by implementing the Runnable Interface:

- We create a new class which implements java.lang.Runnable interface and override run()method.
- Then we instantiate a Thread object and call start() method on this object.

Example:

```
classMultithreadingDemo implements Runnable {
public void run(){
try {
System.out.println("Thread " +Thread.currentThread().getId()+ " is running");
}
catch (Exception e) {
System.out.println("Exception is caught");
}
}
}
class Multithread {
publicstatic void main(String[] args){
```

```
int n = 8; // Number of threads
for (int i = 0; i < n; i++) {
Thread object= new Thread(new MultithreadingDemo());
object.start();
}
}
}
```

Output:

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

Advantages of Multithreading:

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

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

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

Multitasking:

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU.

Multitasking can be achieved in two ways:

1. Process-based Multitasking (Multiprocessing)
2. Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing):

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading):

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

Thread:

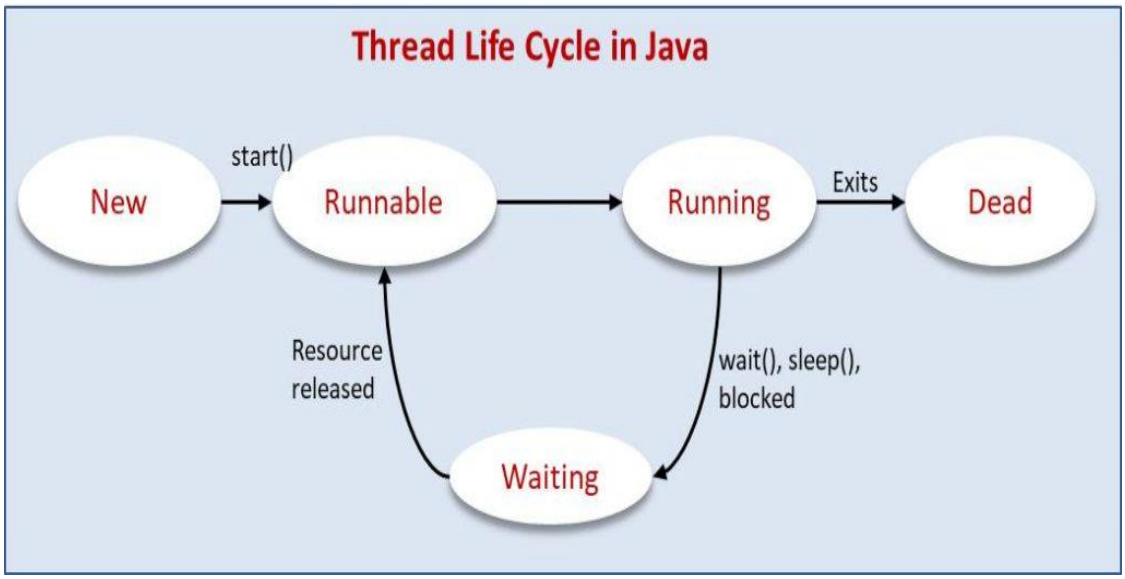
- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Life cycle of a Thread:

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting

5. Terminated



1.New:

- Whenever a new thread is created, it is always in the new state.
- For a thread in the new state, the code has not been run yet and thus has not begun its execution.

2.Runnable:

- A thread, that is ready to run is then moved to the runnable state.
- In the runnable state, the thread may be running or may be ready to run at any given instant of time.

3.Running:

- When the thread gets the CPU, it moves from the runnable to the running state.
- Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

4.Blocked or Waiting:

• Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

5.Terminated or Dead:

A thread reaches the termination state because of the following reasons:

1. When a thread has finished its job, then it exists or terminates normally.

2. It occurs when some unusual events such as an unhandled exception or segmentation fault.

❖ **String handling functions:**

- A string is a collection of characters. In Java, a string is an object that represents a collection of objects.
- A string is a predefined class used to create string objects. It is an immutable object, which means it can't be updated once created.

The string class has a set of built-in-methods, defined below.

- **charAt():** It returns a character at a specified position.
- **equals():** It compares the two given strings and returns a Boolean, that is, True or False.
- **concat():** It Appends one string to the end of another.
- **length():** It Returns the length of a specified string.
- **toLowerCase():** It Converts the string to lowercase letters.
- **toUpperCase():** It Converts the string to uppercase letters.
- **indexOf():** It Returns the first found position of a character.
- **substring():** It Extracts the substring based on index values, passed as an argument.

Example:

```
class Main{
public static void main(String []args)  {
String s1="Adithya";
String s2="Adithya";
String s3="Adi";
```

```
boolean x=s1.equals(s2);
System.out.println("Compare s1 and s2:"+x);
System.out.println("Character at given position is:"+s1.charAt(5));
System.out.println(s1.concat(" the author"));
System.out.println(s1.length());
System.out.println(s1.toLowerCase());
System.out.println(s1.toUpperCase());
System.out.println(s1.indexOf('a'));
System.out.println(s1.substring(0,4));
System.out.println(s1.substring(4));
}
}
```

Output:

```
Compare s1 and s2:true
Character at given position is:y
Adithya the author
7
adithya
ADITHYA
6
Adit
hya
```