# Basics of Java :-

- **Java :-** Java is a high-level, general-purpose, object-oriented and secure programming language developed by Jame Gosling at Sun Microsystems, in 1991. It is formally known as 'OAK'. In 1995, Sun Microsystem changed the name to Java. In 2009, Sun Microsystem take over by Oracle Corporation.

- **Editions of Java :-**

  Each edition of Java has different capabilities. There are three edition of Java :

  i. Java Standard Edition (JSE) : It is used to create programs for a desktop computers.

  ii. Java Enterprice Edition (JEE) : It is used to create large programs that run on servers and manages heavy traffic and complex transactions.

  iii. Java Micro Edition (JME) : It is used to develop applications for small devices such as set-top box, phone & appliences.

- **Features of Java :-**

  - Simple : Java is simple because its syntax is simple, clean and easy to use & understand.
  - Object-Oriented : In Java, everything is in the form of object. A program must have atleast one class and object.
  - Robust : Java makes an effort to check error at runtime and compile time. It uses a strong memory management system called garbage collector. Exception handling and garbage collection features make it strong.
  - Secure : Java is a secure programming language because it has no explicit pointers and the program runs in the virtual Machine. Java contains security manager that defines the access of Java classes.
  - Portable : Java byte code can be carried to any platform.

- **Platform - Independent:** Java provides a guarantee that code Written Once and Run Anywhere (WORA). This byte code is platform independent and can be run on any machine.
- **High Performance:** Java enables high performance with use of the Just In time compiler.
- **Distributed:** Java also has networking facilities. It is designed for the distributed environment of the internet because it support TCP/IP protocol. It can run over internet. EJB and RMI are used to create a distributed system.
- **Multi-threaded:** Java also supports multi-threading. It means to handle more than one job a time.

- **JDK, JRE and JVM :-**

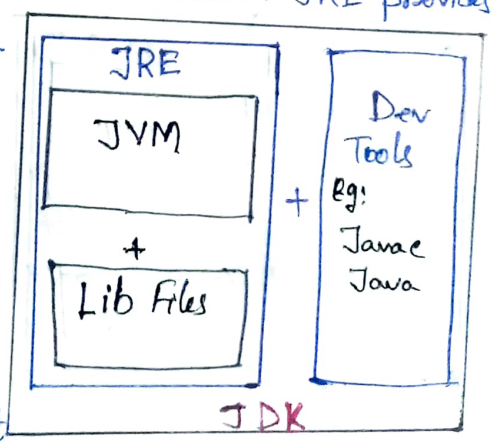
JDK, JRE & JVM are the most important parts of Java programming language.

**JDK :-** JDK stands for Java evelopment Kit. JDK provides an environment to develop and execute the java program. JDK is a kit that include two things,

i) Development tools to provide an environment to develop java programs.

and. ii) JRE to execute Java program

**JRE:** JRE stands for Java Runtime Environment. JRE provides an environment to only run the Java programs onto the machine. JRE is only used by the end users or the system.

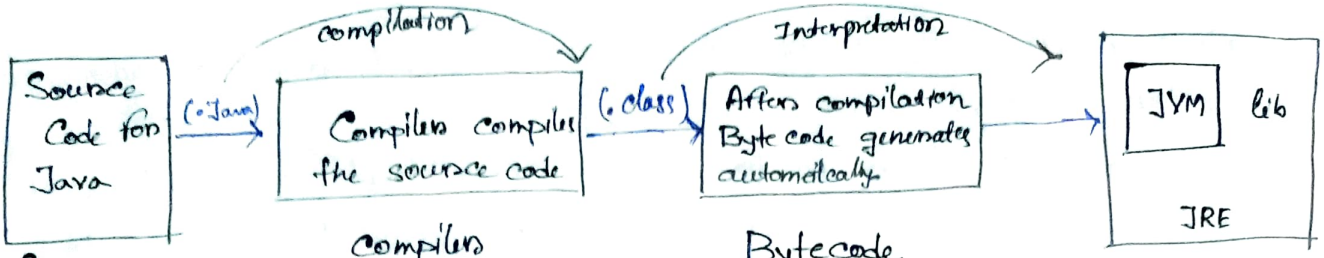



JRE consists of libraries and other files that are uses at runtime.

**JVM:** JVM stands for Java Virtual Machine, which is a very important part of both JDK and JRE because it is inbuilt in both. Whatever Java program we run using JDK and JRE goes into JVM and JVM is responsible for executing the Java program line by line.

## Java Working Flow :-

How does a java code is processed, is shown below by diagram:




**Source Code**

Creating a source code using Java.

**Compiler**

The compiler will compile the source code and will compile fully once there are no error in the code. 'Javac' is the Java compiler.

**Bytecode**

The compiler generates a bytecode and any device which has JVM is capable to run Java, can translate into machine code.

If there is JVM in any system it can read the bytecode and executes it.

## :- Main() Methods in Java :-

The main() method in Java is 'public static void main(String args[])'. Take the knowledge from writing the first code of Java.

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

- **public** : The public is an access modifier that can be used to specify who can access this main() method. It simply defines the visibility of the method. The JVM calls the main() method outside the class. Therefore it is necessary to make Java main() method as public.
- **static** : 'static' is a keyword in Java. The main advantage of static method is that we can call the method without creating an instance of the class. JVM calls the main() method without creating an instance of the class. Therefore it is necessary to make the java main() method static.

- **void** : void is a return type of method. The java main() method doesn't return any value. Therefore it is necessary to have a void return type.

- **main** : main is the name of the method. It is a method where program execution starts.

- **String args[]** : 'String' in java is a class that is used to work on Strings and 'args' is a reference variable that refers to an array of type String. If one wants to pass the argument through the commandline then it is necessary to make the argument of the 'main()' method as 'String args[]'.

---

N.B. :-

(i) The extensions of java code files is ".java"

(ii) The extension of the compiled java classes or bytecode is ".class"

(iii) The command used to compile java code is,

```
>>> javac file_Name.java
```

(iv) The command used to run java code is,

```
>>> java file_Name
```

(v) There are three types of comment in java

i. Single line comment (Eg: // single line comment)

ii. Multi-line comment (Eg: /* This is Multline Comment */)

iii. Document comment (Eg: /** Document Comment */)

(vi) In case of printing a line you use,

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

Here, it will print 'Hello Java' and move to the next line.

## :- Variables in Java :-

- **Variables :** Variable is a name given to a memory location. It is used to store a value that may vary.

  Java is statically typed language, and hence, all the variables are declared before use.

- **Declaration :-** In java, variables can be declared as follows:
  - data_type : Type of data that can be used to store in this variable.
  - name : Name given to the variable.

Syntax : data_type variable_name;
Eg. : int j;

- **Value Assignation :** We can assign the value in this variable by using two ways: i) By Using variable intialization
  ii) By taking input. (Eg: int j = 12)

- **Variable naming Convention in Java :**
  - A variable name should be short and meaningful.
  - It should begin with lowercase letter.
  - It can begin with underscore (_) and dollar ($) sign.
  - If the variable name contains multiple words, then use camel case.
  - Always try to avoid single character variable names such as i, j etc except for the temporary variables.
  - A variable name cannot contain whitespaces.
  - We can't use keywords as variable names.

## :- Java Keywords :-

- **Keywords :** Keywords in Java also known as reserved words. These are predefined words, therefore the can't be used as variable name. If keywords are used as variable name, the result will be compile time error.

The list of all java keyword is given :

There are 48 Keywords in Java:-

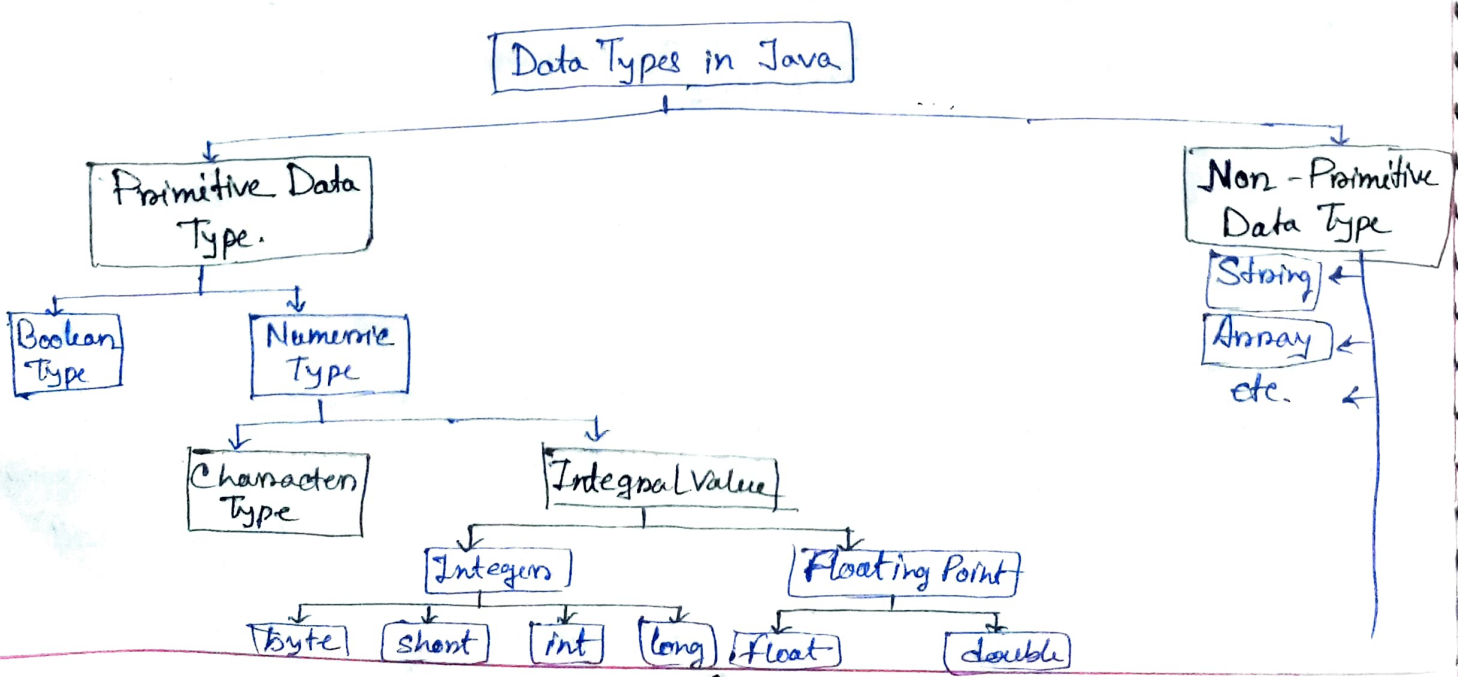
| | | | | |
|---|---|---|---|---|
| abstract | assert | boolean | break | byte |
| catch | char | case | class | continue |
| default | do | double | else | enum |
| extends | final | finally | for | float |
| if | implements | import | instanceOf | int |
| interface | long | native | new | null |
| package | private | static | super | switch |
| synchronized | this | throw | transient | try |
| void | volatile | while | protected | public |
| return | short | throws | | |

## :- Data Types in Java :-

- **Data Type** : The data type defines the type of value that can be stored in a variable

Java has two categories in which data types are segregated.

1. Primitive Data Type.
2. Non-Primitive Data Type or Object Data Type.



1. Primitive Data Type: Primitive data type is predefined by the language and is named by keyword. Primitive datatypes are only single values and have no special capabilities.

There are 8 primitive data types in Java.

i) boolean :- boolean data type specifies only one bit of information and it is used to store only two possible values either 'true' or 'false'.

Syntax: boolean variable_name;

ii) byte :- byte data type is an 8-bit signed two's complement integers. The byte data type is useful for saving memory in large arrays.

Syntax : byte variable_name;

iii) short :- short data type is a 16-bit signed two's complement integer. The short data type is useful for saving memory in large array.

Syntax: short variable_name;

iv) Integer :- It is a 32-bit signed two's complement integer.

syntax : int variable_name;

v) Long :- long data type is 64-bit two's complement integer.

syntax : long variable_name;

vi) Float :- float data type is a single precision 32-bit IEEE 754 floating-point.

syntax : float variable_name;

vii) Double :- double data type is a double precision 64-bit IEEE754 floating-point.

syntax: double variable_name;

viii) Char :- The char data type is used to store characters. The char data type is 16 bit unicode character.

Syntax :- char variable_name;

| Type | Description | Default | Size | Example Literals | Range of Values |
|---|---|---|---|---|---|
| boolean | 'true' or 'false' | false | 1 bit | true, false | true, false |
| byte | 2's complement Integer | 0 | 8 bits (1 byte) | — | -128 to 127 |
| short | 2's complement Integer | 0 | 16 bits (2 bytes) | — | -32768 to 32767 |
| int | 2's complement | 0 | 32 bits (4 bytes) | -2, -1, 0, 1, 2 | -2,147,483,648 to 2,147,483,647 |
| long | 2's complement | 0 | 64 bits (8 bytes) | -2L, -1L, 0L, 1L, 2L | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | IEEE 754 floating point | 0.0 | 32 bits (4 bytes) | 0.3f, 3.14f | upto 7 decimal digits. |
| double | IEEE 754 floating point | 0.0 | 64 bits (8 bytes) | 1.234567e300d, -123456e-300d, 1e1d, | upto 16 decimal digits |

N.B.:- ① By default in java double is considered not the float values. As float has limited precision, whereas double has longer precision and that's why double can have more values.

Now, if you declare a float variable, like 'float x = 3.14;' it does not work, because, by default the value is set as 'double'. So you have to declare the float like, 'float x = 3.14f;'.

② Java follows the 'UNICODE' not the 'ASCII' values.

③ In java characters are enclosed within (' ') single quotes and strings are in(" ")double quotes.

④ In the case of 'long' data type we have to mention a 'l' after intialization, like long var = 5854l;

⑤ Whitening or Implicit Conversion: Automatic conversion of datatypes.

int num = 1.7f;

⑥ Narrowing conversion: Mentioning the typecasting.

2. Non - Primitive Data Types :- Non - primitive data types refers to the objects. The reference data types will contain a memory address of variable values because the reference types won't store the variable directly in the memory.

Eg : Strings, Objects, arrays etc.

↳ String str = "I am learning Java at 15 September".

## Scope of Variables in Java :-

- Scope of Variable :- The variable scope is the part of the program where the variables are accessible. The scope of variable can be determined at compile time.

⊛ There are mainly two types of variable scope:

i) Local Variable Scope : A variable that is defined inside a block, method body or constructor is called local variable. This variables can't be accessed outside the method.

Example -
```
public class VariableScope {
    void method()
    { int x;
    }
}
```

ii) Members / Class level Variable Scope : A variable that is declared inside the class but outside of the method body, block or constructor is known as member/class level variable.

⊛ These variables can be directly accessed anywhere in the class.

Example:
```
class VariableScope {
        int x;
}
public class VariableScopeDemo {
        public static void main (String args[]) {
        VariableScope Obj = new VariableScope();
    } } Obj.x = 10;
```

- **Types of Variable :-** There are three types of variables in Java.

1.) **Local Variable :-** A variable that is defined inside a block, method body or Constructor is called local variable.

Ex
```
public class Addition {
 public void add()
 {
    int a;
    System.out.print(a);
 }
```

ii) **Instance Variable :-** A variable that is declared inside the class but outside of the method body, block or constructor is known as an instance variable.

iii) **Static Variable :** A variable that has been declared as static is known as a static variable. It is also known as class variable

Ex:
```
class Student {
    public static int id;
}
public class StudentDemo {
    public static void main (String args[])
    {
      Student.id = "G2";
    }
}
```

Ex
```
class Student {
  String name;
  int rollno;
}
public class studentrecord
{
  public static void main(
                    String args[])
  { Student obj = new Student();
    Obj.name = "Anik"
    obj.rollno = "G2"
  }
}
```

- Static keyword in Java is used for memory management primarily.
- It can be applied to variables, methods, blocks and nested classes.
- The main concept behind static is that it belongs to the class rather than instance of the class.

- **Type Casting in Java:**

The typecasting is the process or converting one primitive data type to another. It can be done automatically and explicitly.

There are two types of typecasting in java:

i) **Widening or Automatic Type Conversion:**

When we assign a value of a smaller data type to a large data type, this process is known as Widening Type Casting. It is also known as Automatic Type Conversion because the java compiler will perform it automatically. This can happen when two data types are compatible.

The list of compatiblity is,

∴ byte → short → int → long → float → double (Widening Type conversion)

Example: int i = 2147483647

long l = i;

l = l + 1; ⟶ [2147483647]

double d = l; ⟶ [2.147483648E9]

ii) **Narrowing or Explicit Type Conversion:-**

When we assign a value of a large data value to a small data type the process is known as Narrowing Type Casting. This can't be done automatically, we need to convert the type explicitly. If we don't know the perform casting it will give compile time error.

The list of Conversion is:

∴ double → float → long → int → short → byte (Narrowing or Explicit Conversion)

Example:

double d = 25.123

int i = (int) d; ⟶ [25]

byte b = (byte) i; ⟶ [25]

N.B: ① While explicit conversion if the value of the conversion is out of range, then the value will be.

value = value % maximum_range_value ;

Eg:-

```
int a = 257;
byte b = (byte) a;
System.out.println(b);
```

↳ b = 1

Here the value of 'b' will '1'. As the range of byte is 256 (-128 to 127) so,

b = 257 % 256

b = 1

② Char to boolean type casting is not possible.

## :o: Input / Output in Java :o:

- **Print statement in Java:**

i) **Using 'println()' method:** In Java, we usually use println() method to print the text on the console. The text is passed as the parameter to this method in the form of string. Here after printing the statement the cursor move to the start of the nextline & printing starts from next line.

Eg: System.out.println("Java");

ii) **Using 'print()' method:** In Java we usually print with the help of print() method. The method prints the text on the console and after printing text cursor remain at the end of the console. The next printing take place from just here.

Eg: System.out.print("Java");

iii) **Using 'printf()' method:** The 'printf()' method in Java is used to print formatted data on console. 'Print()' & 'println()' method take single arguments, but printf() method take multiple arguments.

Eg: System.out.printf("Formatted precision : PI = %.2f\n", Math.PI");
System.out.printf("%-15s %08d", s, x); → Java.....063

## ■ Taking Input in Java:

In Java, there are mainly two ways to get the input from the user.

↳ Using Scanner Class

↳ Using BufferedReader Class.

▶ **Using Scanner Class:** Scanner is a class in java that is used to take input from the user. It is present in the **java.util**. **Scanner** class is one of the most preferable ways to take input from users. This class is used to read the input of primitive types such as int, double, long etc and String. You need to import Java.util package before scanner class. Once you create and object should be closed (obj.close())

- **Methods of Scanner Class in Java:**

Java scanner class provide various methods to read different primitive data types from the user.

| Method | Description |
|---|---|
| nextInt() | reads 'int' value from the user. |
| nextFloat() | reads 'float' value from the user. |
| nextDouble() | reads 'double' value from the user. |
| nextLong() | reads 'long' value from the user. |
| nextByte() | reads 'Byte' value from the user. |
| nextBoolean() | reads 'boolean' value from the user. |
| nextLine() | reads a line of text from the user. |
| next() | reads a word from the user. |

Eg.
```
import java.util.Scanner;
class Inputs {
    public static void main(String args[]) {
        int a;
        Scanner obj = new Scanner(System.in);
        a = obj.nextInt();
    }
}
```

N.B. to scan a nextLine. you have,
sc.nextLine();
String s = sc.nextLine();

## Using BufferedReader:

It is a simple class that is used to read a sequence of characters. It has a simple function/method that reads a character another read which reads an array of characters and a 'readLine()', which reads a line.

'InputStreamReader()' is a function that converts the 'input stream of bytes' into a 'stream of character'.

BufferedReader can throw checked exceptions. Here we have to 'import java.io.*;

Eg:
```
import java.io.*;
public class Buffered {
    public static void main (String args[])
      throws IOException
    {
      BufferedReader obj = new BufferedReader (new InputS-
      treamReader (System.in));
      String s = obj.readLine();
      System.out.println (s);
    }
```

## Assign Operators in Java:

- In Java '=' is used to assign a value to variable in Right to Left assignation.

  Eg: a = 12 ; [it basically means the value '12' is assigned to the variable 'a'] . a ← 12;]

- Incremental & decremental operators:
  - ↳ Modulus Decrement ⇒ a %= 2 [a = a%2]
  - ↳ Incremental Addition ⇒ a += 2; [a = a+2]
  - ↳ Incremental Multiplication ⇒ a *= 4; [a = a*4]
  - ↳ Decremental Substraction ⇒ a -= 5; [a = a-5]
  - ↳ Decremental Divison ⇒ a /= 2; [a = a/2] [automatic typecast

## ⑥ Increment and Decrement as 'pre-' & 'post-' :-

The increment & decrement are symbolize '++' & '--'

There are two types of increment operations:

i.) Pre-increment : Here the value is firstly incremented & then assigned or fetched as a variable.

Eg:- `int a = 10;` `int b = ++a;` | Here, the value of a = 11, Incremented Value b = 11

ii) Post-increment : Here the vaule is firstly assigned or fetched as a variable, then incremented.

Eg:- `int a = 10;` `int b = a++;` | Here, the value of a = 11, Increment value is b = 10

Similary Pre-decrement & Post-decrement is coceptualize in terms of substraction.

i) Pre-decrement : Here the value is firstly decremented & then assigned or fetched as a vanicable.

Eg:- `int a = 10;` `int b = --a;` | a = 9, b = 9

ii) Post-decrement : Here the value is firstly assigned or feteked as a variable, then decremented.

Eg:- `int a = 10;` `int b = a--;` | a = 9, b = 10

## Arithmetic Operators :-

■ Arithmetic Operators :- Arithmetic Operators in Java are used to perform mathematical operations, ie. Addition, Substraction, Multiplication, Divison and modulus etc.

• The basic anithmetic operators are given below:

(i) Addition Operator (+)
(ii) Substraction Operator (-)
(iii) Multiplication Operator (*)
(iv) Division Operator (/)
(v) Modulus Operator (%)

N.B.: The precedence orden (from highest to lowest) anithmetic operators: %, *, /, +, -

## :- Unary Operators :-

■ **Unary Operators :-** Unary Operators in Java are the types of operators that requires only one operand.

There are various types of Unary operators, Like,

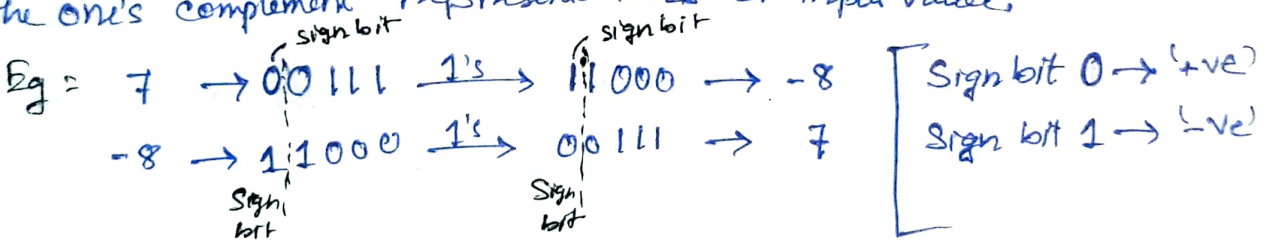
- **Unary minus operator (-):** This operator can be used to convert a negative value into a positive value or a positive value into a negative value.

- **Unary NOT operator (!):** This operator is used to convert true to false or vice versa.

- **Increment operator (++):** → Pre-increment (++i)
  → Post-increment (i++)

- **Decrement Operator (--):** → Pre-decrement (--i)
  → Post-decrement (i--)

- **Bitwise Complement (~):** This operator is used to return the one's complement representations of input values.




## :- Relational Operator in Java :-

The relational operators are used to check the relationship between two operands. This operator is also known as comparison operator because it is used to make comparison between two operands.

There are many types of relational operator, which are given below:

i) **Equal to operator (==):** It is used to check whether the two

operands are equal or not. If they are equal. It returns 'true' otherwise, it return 'false'.

ii) Not Equal to operator (!=) :- This operator is used to check wheather the two number equal or not. If not it returns 'true' otherwise it return 'false'.

iii) Greater than operator (>) : This operator is used to check wheather the first operand is greater than the second operand or not.

iv) Greater than equal to the Operator (>=) : This operator is used to check wheather the first operand is greater or equal to the second operand or not.

v) Less than equal to the operator (<=) : This operator is used to check wheather the first operand is lessen or equal to the second operand or not.

vi) Less than operator (<) : This operator is used to check wheather the first operand is less than the second operand or not.

N:B.:-
- The result of a relational operator is always 'boolean'.
- There are required minimum 2 operand to compare or use relational operator.

## :- Logical Operator in Java :-

- **Logical Operator :-** These operator are used to perform operation such as 'OR', 'AND', 'NOT'. It operates on two boolean values, which return true or false as a result. There are three types of Logical Operator in Java:
  - (i) Logical AND operator (&&) ⇒ boolean_1 && boolean_2
  - (ii) Logical OR operator (||) ⇒ boolean_1 || boolean_2
  - (iii) Logical NOT operator (!) ⇒ boolean_1 != boolean_2

## :- Bitwise Operator in Java :-

- **Bitwise Operator :-** The bitwise operator are used to perform bit manipulation on numbers.
  - **Bitwise AND operator (&) :** It takes two numbers as operands and does AND on every bit of two numbers.

Eg a = 6
b = 7
a & b ⇒ 1 1 0 / 1 1 1 → 6 ∴ a & b = 6

  - **Bitwise OR operator (|) :** It takes two numbers as operands and does OR on every bit of two numbers.

Eg: a = 6
b = 7
a | b = 1 1 0 / 1 1 1 → 7 ∴ a | b = 7

  - **Bitwise NOT operator (~) :** It takes one number and inverts all the bits.

Eg: a = 5 ; ~a = [00101 → 11010] = -2.

- Bitwise XOR operator (^) : It takes two numbers as operand and does XOR on every bit of two numbers.

Eg: a = 6 — 110

b = 7 — 111

a ^ b — 001 = 1

- Left Shift Operator (<<) : It takes two numbers, the left shift operator shifts the bit of the first operand, the second operand decides the number of places to shift.

Eg: int a = 8
a << 2
∴ a = 32

8 ⇒ 1000
a << 2 ⇒ 100000 → 32

- Right Shift Operator (>>) : It takes two numbers, the right shift operator shifts the bit of the first operand, the second operand decides the number of places to shift.

Eg: int a = 8
a >> 2
a = 2

a = 1000
a >> 2 = 10

## :- Ternary Operator in Java :-

- Ternary Operator :- Java ternary operator is also known as conditional operator : This operator is the only conditional operator in Java that takes three operands. We can use ternary operator in place of if-else statement or even switch statement.
  - Syntax : variable = condition ? true_expression : false_expression ;

## Instance of Operator :-

The instance of operator in Java is used to compare an object to a specified type. We can use it to check if an object is an instance of a class, an instance of subclass, or an instance of a class that implement a particular interface.

The instanceof operator is either returned true or false.

## :Control Statement in Java:-

■ Control Statement : A control statement is used to control the flow or the execution of a program. In Java, we can control the flow the flow of execution of a program based on some conditions.

In Java, we can put control statement in the following three categories. These are 'selection statement', 'iteration statement' and 'Jump Statements'.

1. Selection Statements:- The selection statement allows your Java program to choose a different path of execution based on certain condition.

i) if statement : The if statement in Java is a decision making statement that determines wheather or not a certain statement or block of statement will be executed.

The block of statements is executed if a certain condition evaluates to true; otherwise, it not executed.

• Syntax:
```
if(condition)
{
// code is executed if the condition is true.
}
```

ii) if-else statement : Java 'if' statement is used to decide wheather a certain statement or block of statements will be executed or not. Now, if the 'if' condition is false then the use 'else' statement helps to occur the code when the condition false.

• Syntax:
```
if(condition)
{
// executes this block when the condition is true
}
else
{
// executes this block when the condition is false
}
```

Eg:-

```
public class IfElseStatement {
    public static void main(String args[])
    {
        int a = 46
        if (a%2 == 0)
        {
            System.out.println("Even Number");
        }
        else
        {
            System.out.println("Odd Number");
        }
    }
}
```

iii) nested if-statement: nested if-statement means if statement inside the another if statement. In nested-if statements, the inner if block statements executes only if the outer block of statement is true.

- Syntax:

```
if (condition-1)
{ //executes of this block if codition 1 is true
    if (condition-2)
    {
        //executes this block if condition 2 is true.
    }
}
```

iv) if-else-if ladder: When we need to deal with different conditions in Java, we use if-else-if ladder. Form the top-down, the if statements are executed..

The assumption connected with that it is executed as soon as one of the conditions governing the 'if' is true, and the reminders of the ladder is bypassed. The final 'else' statement will be executed if none of the conditions are valid. The final else sentence serves as default condition.

- Syntax:

```
if ( condition1 )
{
 // Condition_1 true & then executes
}
else if (Condition_2)
 {
 // Condition_2 true & then executes
 }
 else if (Condition_3)
 {
 // Condition_3 true & then executes
 }
 ...
 else
 {
 // If all the condition is false then this part is executed.
 }
```

v) Switch Statement:

A multiway branch statement in Java is the switch statement. It can be used to run a single statement to run a single statement based on a set of conditions. It will deal with datatypes such as 'byte', 'short', 'char' and 'int'. The default statement is used if none of the constants fit the value of expression. It is not necessary to use the default statement. The declaration sequence is terminated with the split statement with in the term event.

- Syntax:

```
switch (expression) {
    case constant_1 :
        ...
        break;
    case constant_2 :
        ...
        break;
    default :
        ...
}
```

N.B: Various Syntaxes of Switch case;

(i) switch (Exp) {
case 1, 2: ...
break;
case 3, 4: ...
break;
case 5: ...
break;
default: ...

(ii) Without Break by using arrow.

```
Switch (Exp){
case 1 -> operation1;
case 2 -> Operation 2;
case 3 -> operation 3;
default -> operation_final;
```

iii) Without breaks without arrow;

```
Switch (Exp){
case 1 : yield operation 1;
case 2 : yield operation 2;
case 3 : yield operation 3;
default : yield operation final;
```

## 2. Iteration Statements :-

Java iteration statements are used to repeat the set of statements until the condition of the termination is true.

There are three types of iteration statements in Java.

:-

(i) While loop: Java while loop is used to repeat a statement or block of the statement until a condition is true. "We can use a while loop if the number of iteration is not fixed". The condition of a while loop is any boolean expression.

The loop will run till the condition is true, if the condition is false the control goes to the next statement immediately following loop.

- Syntax:

```
initialization;
while (condition){
    // statement
    updatation_expression;
}
```

Eg:-
```
public class JavaWhileLoop {
    public static void main (String args[])
    {
        int i = 1;
        while (i<=10) {
            System.out.print (i + " ");
            i++;
        }
    }
}
```

(ii) **do-while loop**: Java do-while loop is also used to repeat a statement untill a condition is true. Sometimes in our program we want to execute the body of the loop atleast even if the condition expression is false. Then we should go for do-while loops in Java.

It executes the body loop atleast once because the condition expression is checking at the end.

**Syntax:**

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

Eg:
```
public class DoWhileLoop {
    public static void main (String args[])
    { int i = 1;
        do {
            System.out.println (i + " ");
            i++;
        } while (i<=10);
    }
```

iii) for Loop : The for loop in Java is used to iterate a part of the program several. It consumes the initialization, condition checking and increment/decrement a value in one line. If the number of iterations is fixed then it is recommended you use for loop.

Syntax: for ( initialize ; condition checking ; increment/decrement)
{
// Code_to_be_executed
}

Example :
```
public class JavaForLoop {
    public static void main (String args[]) {
        for ( int i=0; i<=10; i++)
        {
            System.out.print( i + "  ");
        }
    }
}
```

N.B. : Java Enhanced for Loop : Java Enhanced for Loop provides a simpler way to iterate through the elements of a collection of array. It is used when we sequentially iterate through elements without knowing the index of the currently processing element. • Syntax :- for (T element : collection obj/arr) { }

Ex :
```
public  JavaEnhancedForLoop {
    public static void main (String args[]) {
        String array [] = { "I", "am", "Anik", "Pal"};
        for (String x : array) {
            System.out.println (x);
        }
    }
}
```

3. Jump Statements : In Java, we use jump statement to shift the control of the program to other parts of the program. There are various types of jump statements in Java

i) break statement :- The loops are terminated using Java break expression. It can be used within a loop. When a break statement is found with in a loop, the loop is ended and control is passed to the next statement after the loop.

- Syntax : break;

ii) labbled break statement :- Java lebelled break statement can be used as a civilized form of a goto statement.

Eg
```
first : {
        second {
                System.out.println("Hola");
                third {
                        break second;
                }
        }
}
```

iii) Continue Statement :- Java continue statement is used to skip the current iteration of the loop.

- Syntax : continue;

iv) return Statement : Java return statement is used to explicitly return from method. It transfer program control back to the caller method. The return statement immediately terminates the method in which it is executed.

# Methods in Java

■ **Methods :-** The 'methods in Java' or 'methods of Java' is a collection of statement that perform some specific task and return the result to the caller or without returning anything.

Java methods allows us to reuse the code without retyping the code.

- In Java, every methods must be a part of some class (it may be defined in the same class where the main is defined / external class.)
- A method is used to expose the behaviours of an object.

Syntax :- `<access_modifiers> <return_type> <method_name> (list_of_parameters)`

```
{
    // body
}
```

## • Method Declaration :-

i) **Modifier :** It defines the access type of the method i.e. from where it can be accessed in your application. In Java there are 4 types of access specifiers. (Though it is optional)

- **public :** It is accessible to all classes in your application.
- **private :** It is accessible only within the class in which it is defined.
- **protected :** It is accessible with in the class it is defined and in its subclasses.
- **default :** It is declared / defined without using any modifiers. It is accessible within the same class and the package within which a class is defined.

ii) The return type: The data type of the value returned by the method or void (if does not return a value). It is mandatory in syntax.

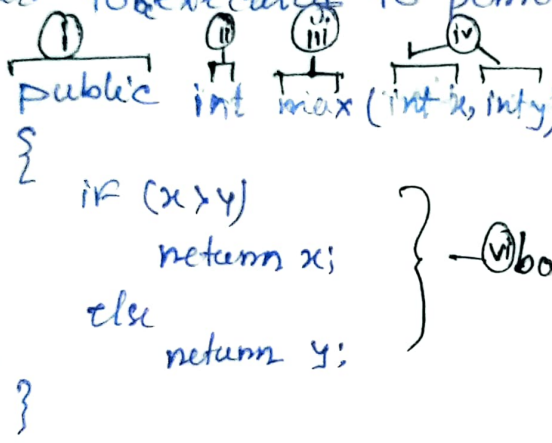
iii) Method Name: The rules for field names apply to method names as well, but the convention is little different.

iv) Parameter List: Comma-separated list of the input parameters is defined, preceded by their data type, within enclosed parentheses. If there are no parameters, you must use empty parentheses ().

v) Exception list: The exceptions you expect by the method can throw, you can specify exception(s).

vi) Method body (): It is enclosed between braces. The code you need to be executed to perform your intended operations.

Example:




- Types of Method: There are two types of method,

i) Predefined Method: In Java predefined methods are the methods that are already defined in Java class libraries is known as predefined methods.

We can directly use these methods just by calling them in the program at any point. (Eg: Math.sqrt(); Math.round();)

ii) User-defined Method: The method written by the user or programmer is known as a user defined method. These methods are modified according to the requirement. There are 4 types of UDM:

- No Argument and No Return Value
- No Argument passed but return a value.
- Argument passed but do not return a value.
- Argument passed and return a value

- **Way of Creating Methods:**

There are two ways to creating a method. in Java.

i) **Instance Method:** Access the instance data using the object name, Declared inside a class.

Syntax:
```
void method_name()
{
  //body.
}
```

ii) **Static Method:** Access the static data using class name, declared inside class with static keyword.

Syntax:
```
static void method_name()
{
  //body
}
```

- **Advantages of Methods:**
  - Reusablity of Code
  - Abstraction (Hiding complexity)
  - Improved Readablity
  - Encapsulation
  - Improved modularity, testability, performance.

- **Passing Parameters to Methods:**

The formal parameters are allocated to a new memory when a parameter passed to a method using the passed by value method. The value of the's parameters is the same as the value of actual parameters..

Changes in formal parameters would not represent changes in individual parameters because they are assigned to a new memory.

# Method Overloading:-

■ **Method Overloading:-** Method overloading in Java is when a class has multiple methods of the same name but different parameters.

The main advantage of the method overloading is to increase the readability of the program. Method overloading is related to compile-time polymorphism.

- **Ways to overload a method:**

i) → By changing the numbers of arguments.

ii) → By changing the data type.

```
Eg: Class Addition {
        public int add (int num1, int num2)
        {
            return num1+num2+num3;
        }
        public int add (int num3, int num2, int num3)
        {
            return num1+num2+num3;
        }
        public double add (double num1, double num2)
        {
            return num1+num2
        }
    }
public class MethodOverloading {
    public static void main (String Args[])
    { double result2;
      int result, result1;
      Addition obj = new Addition(); // Object Creation
      result = obj.add (10, 20);
      result1 = obj.add (10, 20, 70);
      result2 = obj.add (3.14, 8.98, 8.17);
      System.out.print( result + " " + result1 + " " + result2);
    }
}
```

Change of numbers or parameters

Change of data types.

# Memory Allocation

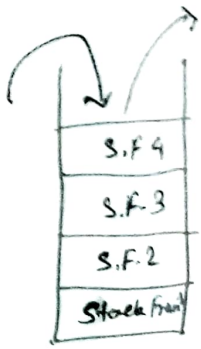
- **Memory Allocation:** Memory allocation in Java specifies the mechanism where the computer program and services are assigned dedicated virtual memory spaces.

The Java Virtual Machine splits the memory into **Stack & Heap Memory.**

- **Stack Memory:** In Java, Stack Memory for static memory allocation and thread execution. Methods, local variables and reference variables are all stored in the Stack portion of memory.

Since they are interpreted in a Last-In-First-Out way, the values in this memory are temporary and restricted to particular methods. When a new method is created, a new block is generated on top of the stack that includes values specific to that method, such as primitive variable and object references.

When a method completes its execution the resulting stack frame is cleared, the flow returns to the calling method, and space for the next method becomes available.




- **Keypoints:**
  - It expands and contracts as new methods are called & returned.
  - Variables within the stack only last as long as the method's scope remains.
  - When the method is executed, it is properly allocated & deallocated.
  - Java throws `java.lang.StackOverFlowError` if the memory is full.

- Since each thread runs in its own stack, this memory is thread-safe.
- As compared to heap memory, access to this memory is fast.

## • Heap Memory :

Any time an object is created and allocated in Java Heap Space, it is used. In heap memory, new objects are often formed, and references to these objects are stored in stack memory. Garbage Collection, a discrete function, keeps clearing the memory used by previous objects that have no reference. A heap space object can have unrestricted access throughout the program.

These objects are accessible from anywhere in the program and have global access.


### Generations of Memory Models:

↳ **Young Generation:-** All new objects are allocated to and aged in this memory.

↳ **Old or Tenured Generation:** This is the memory where long-lasting items are kept. When objects are stone in the young generation, an age threshold is set, and when that threshold is met, the object is transferred to the old generation.

↳ **Permanent Generation:** This is a collection of JVM metadata for runtime classes & application method.

## • Keypoints :

- Complex memory storage methods such as young, old or tenured and permanent generation. are used to access it.
- Java, throws Java.lang.OutOfMemoryError. if the heap memory is full.
- Access to this memory is slower than the stack.
- Unlike stack, this memory is not immediately deallocated. Garbage collecton is used to free up unused objects to maintain memory effic
- A heap, unlike a stack, is not thread-safe and must be protected by synchronizing the code property.

# Garbage Collection in Java:

- **Garbage Collection:** Garbage collection in Java is a process of destroying runtime unused object. Garbage collectors destroy the objects automatically. A garbage collector's key goal is to allow effective use of memory.

- **Ways to make an object eligible for the garbage collector:-**

  There are primarily three ways to make an object eligible for garbage collection.

  ↳ i) Nullifying the reference variable:

```
Student obj = new Student();
obj = null;
```

  ↳ ii) By assigning a reference variable to another:

```
Student obj1 = new Student();
Student obj2 = new Student();
obj1 = obj2;
```

  ↳ iii) By anonymous object:

```
new Student();
```

- **Ways for requesting JVM to run garbage collector:-**

  There are two ways for requesting a JVM to run a garbage collector.

  - Using `System.gc()` method.
  - Using `Runtime.getRuntime().gc()` method.

```
Test t1 = new Test();
Test t2 = new Test();
t1 = null;
System.gc();
t2 = null;
Runtime.getRuntime().gc();
```

N.B:- Garbage collection cannot be controlled by a program. A program can request to run a garbage collector.

- Firstly you have to make an object eligible for garbage collector & then requesting a garbage collector.

# Arrays in Java

- **Array:-** A collection of elements of the same data type placed in contiguous memory locations that can be accessed randomly using an array's indices is called an array.

They can store collection of primitive data-types such as an int, float, double, char, etc., of any particular data type.

Instead of declaring different values to different variables, it is used to store several values in a single variable.

- **Significance of Array:-**

We can use variables like a, b, c, temp etc when we have small number of objects. However, if we need to store a large number of instance, managing them with a regular variables becomes difficult. An array's purpose is to represent multiple instances in a single variable.

Using arrays saves us from time and effort required to declare each element of the array individually.

[N.B.:- In Java Array is used as an object]

- **Declaration of Array:**

To use an array in a program, we must declare a variable to refer to the array and specify the form of the array the variable may represent. (N.B. - It cannot be modified once specified).

- **Syntax:** (i) `data_type [ ] array_reference_variable;` (Preferred)
  (ii) `data_type array_reference_variable[ ];`

- **Eg:** (i) `int [] arr;`
  (ii) `int arr[];`

- **Creating An Array:-**

Declaring an array variable does not create an array. We have to create it after declaration.

- **Syntax:** `array_reference_variable = new data_type [array_size];`

Ex: `int [] ann;` //Declaration

`ann = new int[4];` // Creation

The above statement does two things:

(i) It creates an array using the "new" keyword.

(ii) It assigns the reference of the newly created array to the reference variable of array.

N.B: The declaration of an array variable, the creation of array and the assignment of the array's relation to the variable can all be done in one statement. The Syntax is given below.

- Syntax: `data_type [] array_ref_variable = new data_type[array_size]`
- Eg: `int[] ann = new int[10];`

## ■Intializing an Array:-

There are multiple ways to intialize—

i) **In Single Line:-** The syntax for creating and intializing an array in a single line as follows:

Syntax: `dataType [] array_Ref_variable = {value0, value1, ..., valueN};`

Eg: `int[] ann = {12, 8, 2004};`

ii) **Using Loop:** An array can be intialized using a loop.

Eg:-

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

class Loop{
    public static void main (String args[]){
    int[] ann = new int[5];
    Scanner sc = new Scanner(System.in);
    for (int i=0; i<5; i++){
    }}     ann[i] = sc.nextInt();}
```

- **Default values for different data-types:**

All the arrays' elements after creating arrays are initialized to default values, if we don't initialize them while creating.

| Data type | Default Values |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0 |
| char | '\u0000' |
| String | null |
| boolean | false |

## ■ Accessing Elements of an Array:

An element of the array could be accessed using indices, and indexing starts from '0'. So, if there are 'n' number of elements are present, the first index will be '0' & the the last one will be 'n-1'; so the range will be '0' to 'n-1'.

Syntax: Array_name [index];
Eg. : arr [3];

## ■ Array Storing in Java:

Array in java store either of the two things:

(i) Either primitive values or references (a.k.a pointers). When you use "new" command to build an object, memory is reserved on the heap and a reference is returned.

(ii) Reassigning references & garbage collector. All the reference variables can be reassigned repeatedly, but their data type to whom they will refer is fixed at the time of their declaration.

## ■ Passing Arrays to a Function / Method:

- **Passing Array as a function parameter:-** In Java programming language, the parameter passing is always made by value. Whenever we create a variable of a type, we pass a copy of its value to the method.

▸ **Passing Reference Type** – We store reference of non-primitive data types and access them via references. So in such cases, the references of Non-Primitive data types are passed to a function/method.

Eg:
```
public class Solution{
    public static void main (String[] args){
int[] arr = {1,2,3,4,5};
print(arr);   //Reference to array is passed
}
public static void print(int[] arr) {
for (int i=0 ; i<5 ; i++)
{
System.out.print (arr[i]+" ");
}
}
```

▸ **Returning Array from a Method:**

Similarly, as we pass reference as a function parameter we will return reference too in case of array.

Eg:
```
Public class Arrayuse {
    public static void main (String[] args) {
        int A[] = inputElements();
        for (int i : A) {
            System.out.println(i)
        }
}
Public static int[] inputElements() {
    int[] arr = new int[2];
    arr[0] = 6;
    arr[1] = 12;
    return arr; } }
```

## Two dimensional Array:

In Java, we can define multidimensional arrays, in a simple words as an array of arrays. A two-dimensional array is the simplest kind of multidimensional array.

A list of one dimensional arrays makes up a two dimensional array.

- Syntax to declare a 2-D array of size xy:

```
int [][] twoD_array = new int[x][y];
```

Here 'x' is the row & 'y' is the column number. A two dimensional array can be seen as a table of 'x' row & 'y' columns, where the row number ranges from '0' to 'x-1' & column number ranges from '0' to 'y-1'.

- Intialization of two-dimensional array:

↳ Indirect Method of Initialization:

Syntax: `array_name [row_index] [column_index] = value;`

Eg.: `arr [0] [0] = 11;` [N.B.: To Intialize an 2-D array Row specify is necessary.

↳ Direct Method of Initialization:

Syntax:
```
data_type[][] array_name = { { value R1C1, value R1C2, ... }
                             { value R2C1, value R2C2 ... }
                           };
```

Eg:
```
int [][] mat = { {0, 1, 2, 3}
                 {4, 5, 6, 7}
                 {7, 8, 9, 10}
               };
```

There are three rows and four columns in the following series. The element of the braces are stored in a table from left to right as well. The array would be filled in the order of the first four elements from the left in the first row, the next four elements in the 2nd row & so on.

- **Accessing two-dimensional Array elements:-**

An element in the 2-dimensional array is accessing using the subscript, i.e. 'row index' & 'column index' of the array.

**Syntax:** array_name [row_index][column_index];

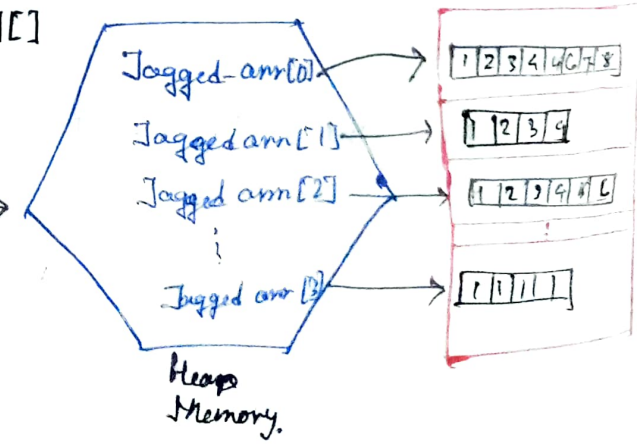
**Eg:**
```
int [][] arr = { {2, 1}
                 {0, 4}
                 {8, 1}};

for (int i=0 ; i<3 ; i++)
  {
    for(int j=0; j<2; j++)
      {
        S.O.U.T (arr[i][j]);
      }
  }
```

## (*) Jagged Array in Java:-

- **Jagged Array:** A jagged array is an array of arrays such that members can be of different sizes, i.e. we can create a 2-D array but the variables number of columns in each row. This type of are also known as jagged array,

- **Syntax:**

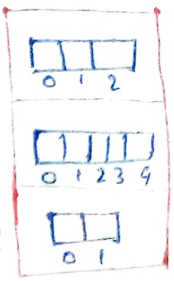
```
data_type array_name[][] =
        new datatype[n][];
array_name[] = new data_type[n1];
array_name[] = new data_type[n2];
array_name[] = new data_type[n3];
```




[n1 ⇒ No. of column in row 1
n2 ⇒ " " " " row 2

• Others way to implement or Intialize Jagged Array :-

(i)
```
int arr = new int [3] [ ]; // Declaring 2-D array with 3 rows.
arr[0] = new int [3]; // First row has 3 column
arr[1] = new int [4]; // Second row has 5 column
arr[2] = new int [2]; // third row has 2 column
```




(ii)
```
/* Taking random input */
for (int r=0 ; r<arr.length ; r++)
  {
    for (int c=0; c<arr[r].length ; c++)
     {
       arr[r][c] = (int) (Math.random()*10);
     }
  }
```

(iii)
```
/* Printing Output */
for (int i=0 ; r<arr.length; r++)
 {
   for ( int c=0 ; c<arr[r].length; c++)
    {
     System.out.print(arr[r][c]+" ");
    }
    System.out.println()
 }
```

```
for ( int [] r: arr)
  {
    for (int c : r)
     {
       System.out.print(c+" ");
     }
     Sout.ln()
  }
```

(*) Disadvantage :

(i) Once you create an array size you can change it.

(ii) If you want to store an extra element when the array is ~~stat~~ saturated with all the elements you have to create a new array & copy all the elements then change or update the value.

## ✱ Array of Objects:

We can create an array where every array element is a object's reference variable

**Example:**

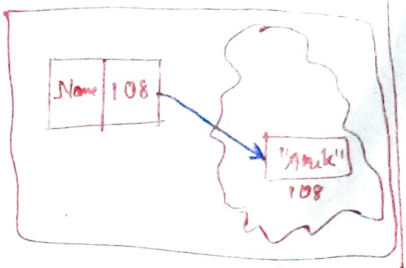
```
class Details {
    int id;
    String name;
    int mark;
}

class Student {
public static void main(String[] anik)
 { Details S1 = new Details();
     S1.id = 10;
     S1.name = "Anik";
     S1.mark = 60;
   Details S2 = new Details();
     S2.name = "Rana";
   Details S3 = new Details();
     S3.name = "Sumon";

   Details[] arr = new Details[3];
     arr[0] = S1;   // Inserting S1 object as first array element
     arr[1] = S2;   // Inserting S2 " " Second element
     arr[3] = S3;   //  "   S3  "  " third "

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

`arr[i].name` ← Use as object's Ref. Variable.

# Strings In Java

- **Strings:** Strings are a sequence of characters. In Java, strings are the objects. The java platform provides the String class to create and manipulate strings.
  - Explicit Construction: `new String variable = new String("data");`
  - Normal Syntax: `String variable = "String data";`

- **Construction:-**
  (i) By using '+' operator between 2 string we can concatenate two string:




Eg:
```
String Name = "Anik";
System.out.print("Hello "+Name);
```

(ii) By using "concat()" method after the specified word. we can concatenate.

Eg:
```
String Name = "Anik";
System.out.print(Name.concat("Pal"));
```

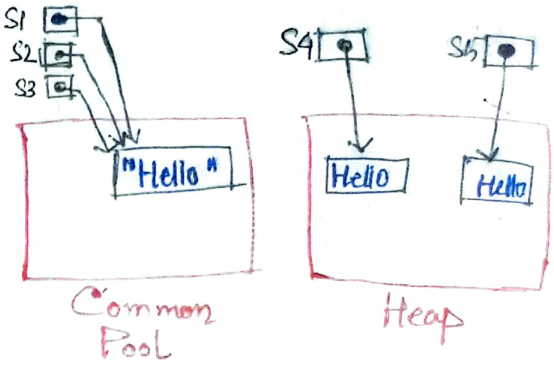
N.B:- We can create a char array and treat them as a string. So, an array of character works the same as a Java string.

Eg:
```
public class HelloWorld{
    public static void main(String[] anik)
    {
        char [] ch = {'H','E','L','L','O',' ',' ','A','N','I','K'};
        System.out.println(ch);
    }
}
```

✱ **Accessing the string elements:** We can access the characters in a string using the `charAt()` method returns the characters at the specified index in a string. It throws IndexOutOfBounds-Exception if the index value passed in the 'charAt()' method is less than '0' or greater than or equal to the string length. Syntax: `String_name.charAt(Index_number)`

## String Literals Vs String Objects:

```
String S1 = "Hello";
String S2 = "Hello";
String S3 = S1;
String S4 = new String("Hello");
String S5 = new String("Hello");
```




Java has provided a special mechanism for keeping the String literals - in a so called 'String common pool'. If two string literals have same contents, they will share the same storage inside the common pool. This approach is adapted to conserve storage for frequently used string."

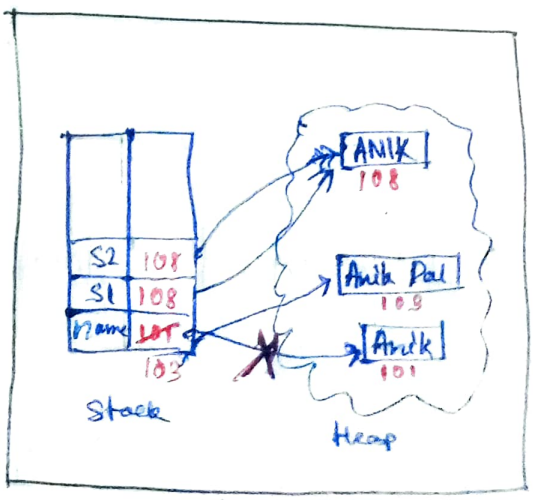
On the other hand, string objects created via the new operator and constructor kept in heap memory. Each string object in a heap has its storage just like an other object. "There is no sharing of storage in the heap even if two string objects have the same content."

N.B:- charAt() method of the string is used to obtain character at specified index.

## Mutability of String:

Since string Literals with the same contents share storage in the common pool, Java's string is designed to be Immutable. That is, once a string is constructed, its content cannot be modified. Otherwise, the other string references sharing same storage location will be affected by the change, which can be unpredictable & undesirable.

```
Public class Immutable {
    public static void main (String a[])
    {
        String name = "Anik";
        name = name + "Pal";
        String S1 = "ANIK";
        String S2 = "ANIk";
    }
}
```



## StringBuffer and StringBuilder:

⊛ **StringBuffer:-** We have seen that a Java String is immutable. A StringBuffer is similar to a string but with more functionality.

A StringBuffer is mutable, and its length and subsequent content can be modified at any time by using specific method calls.

Multiple threads can be used in StringBuffer safely. The methods are synchronized as required such that all operations on any specific instance behave as if they occured in any sequential order consistent with the order of method calls made by any of the individual threads involved.

▸ **StringBuffer Class Construction:**

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

  It creates an empty string buffer with an initial capasity of 16 characters.

- StringBuffer(int capacity): `StringBuffer sb1 = new StringBuffer(25);`

  It creates an empty string buffer with the specified initial capasity (Eg- Here it is 25).

- StringBuffer(CharSequence Seq):

```
CharSequence seq = "Hello Java Strings";
StringBuffer sb2 = new StringBuffer(seq);
```

It creates a string buffer with the same characters as specified CharSequence.

- StringBuffer(String str):

```
String str = "Hello Java";
StringBuffer sb3 = new StringBuffer(str);
```

It creates a string buffer that is initialized by the contents of the specified string.

## ✳ StringBuilder:

StringBuilder, an equivalent class, built for use by a single thread to JDK 5's StringBuffer class.

The StringBuilder class is consistent with StringBuffer, but no synchronization is guaranteed. This class is intended to be used as a drop-in substitute for StringBuffer in places where a single thread previously used in string buffers.

### ▸ StringBuilder Class Construction:

All the things are similar to the StringBuffer();

Every StringBuffer and StringBuilder has a capacity. It is not necessary to assign a new internal Buffer array as long as the length of the character (char) sequence in the StringBuffer or StringBuilder does not surpass the capacity. When internal buffer overflows, it is automatically increased in size.

N.B:- • It is recommended that the StringBuilder class can be used instead of StringBuffer whenever possible, as it is faster in most implementations.

• StringBuilder instances, on the other hand, are not secure to use by multiple threads. If such synchronization is needed, StringBuffer is the best choice.

```
String s1 = new String ("Hello");
String s2 = new String ("World");
System.out.println(s1 = s2); ⟹ World
```

• equals() is a method of a class String, it is used to check equality of two string or objects. If they are equal true is returned else false.

## :- Important Methods of Java :-

- **String "Length" method :** String class provides `length()` method to determine the length of the Java string. It return the number of characters contained in the string object.

  Eg: String str = "Anik Pal";
  System.out.print (str.length());

- **String "indexOf" method :** String class provides `indexOf()` method to get the index of a character in Java string. It returns the index with in this string or the first occurrence of the specified characters.

  Eg: String str1 = "ANIK PAL";
  System.out.println(str1.indexOf('A'));
  System.out.print(str1.indexOf('P'));

- **String "contains" Method :** String class provides the `contains()` method to check if a string contains another string (sequence of character value). It returns true if and only if the string contains the specified sequence of char value.

  Eg: String str2 = "Hello Java";
  System.out.print (str2.contains ("Java"));

- **String "endsWith" method:** String class provides `endsWith()` method to check if the string ends with the specified suffix (sequence of char/string) or not.

  Eg: String str3 = "Coder";
  System.out.println(str3.endsWith("er"));

- **String Java "toLowerCase" & Java "toUpperCase" method :** String class provides `toLowerCase()` and `toUpperCase()` methods to convert string to lowercase & Uppercase characters, respectively.

  Eg: String str4 = aNik paL;
  System.out.println (str4.toLowerCase());
  System.out.print (str4.toUpperCase());

# Object Oriented Programming (OOPs in Java)

- **Object Oriented Programming (OOPs) :-** OOPs as the term implies, refers to a programming language that utilizes the concepts of class & object. It aims to implement real-world entities like inheritence hiding, poly-morphism etc. in programming.

- **Advantages of OOPs :-**
  - OOPs make the developement & maintence of projects easier.
  - OOPs provide the features of data hiding that is good for security concern.
  - We can provide the solution of real world problems if we use object oriented programming.

- **OOPs Concepts :**
  - Packages
  - Access Modifiers
  - Class
  - Object
  - Constructor
  - Inheritence
  - Polymorphism
  - Encapsulation
  - Abstraction.

## :- Classes in Java :-

- **Class :-** A class in Java is a set of objects which shares common characteristics and common properties. It is a user-defined prototype / template / blue-print from which objects are created.

- **Properties of Java Classes :-**

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a groups of methods.

4. A class in Java can contain:
   - Data Member
   - Method
   - Constructor
   - Nested Class
   - Interface.

## ■ Class Declaration in Java :

```
access_modifier  class  class_name
{
    data_Member ;
    method ;
    Constructor;
    Nested class;
    interface;
}
```

Eg:-
```
public class ClassConcept {
    public static void main( String[] args) {
        Student obj = new Student ( );
        obj.roll = 62 ;
        obj.name = "Anik Pal";
        obj.mark = 85.10f;
        obj.nobj.iq = 10;
        obj.nobj.CritThink = "Great";
        System.out.println ( obj.roll + obj.name + obj.mark
                             + obj.nobj.iq + obj.nobj.CritThink);
    }
}
class Student {
    int roll ;
    String Name;
    float mark;
    Inteligence nobj = new Inteligence ( ); // Nested Class's Object
    class Inteligence {          // Nested Class.
        int iq ;
        String CritThink; }
}
```

■ Components of Java Classes:

In general, class declaration can include these components, in order:

i) Modifiers: A class can be public or has default access.

ii) Class Keyword: class keyword is used to create a class.

iii) Class name: The name should begin with an initial letter (capitalized by convention).

iv) Superclass (If any): The name of the class's parent (superclass), if any preceded by the keyword extends. A class can only extend (subclass) one parent.

v) Interfaces (If any): A comma-separated list of interfaces implemented by class, if any, preceded by the keyword implements. A class can implement more than one interface.

vi) Body: The class body is surrounded by braces, {}.

N.B: Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behaviour of the class and its object.

There are various types of classes that are used for real time application such as nested classes, anonymous classes, and lambda expression.

## Java Object

■ Object: An object in java is a basic unit of object oriented programming and represents real life entities.

Objects are the instances of a class that are created to use the attributes and methods of a class.

A typical Java program creates many object, which as you know, interact by invoking methods.

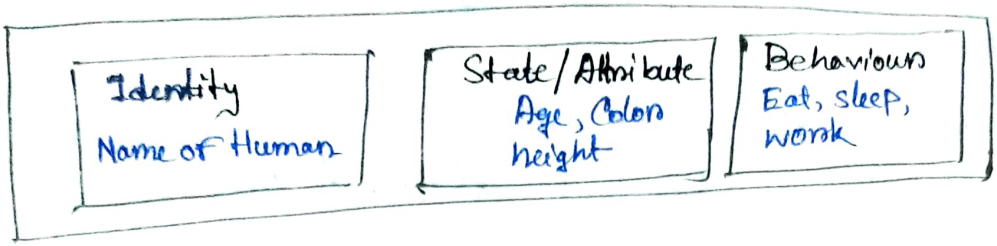
■ An object consists of:

1. State: It is represented by attributes of an object. It also reflects the properties of an object.

2. Behaviours : It is represented by the method or an object. It also reflects response or an object with other object.

3. Identity : It gives a unique name to an object and enables one object to interact with other objects.

- Example of an Object : HUMAN




- Ways to Create an Object of a Class :

There are 4 ways to create object in Java. Strictly speaking, there is only one way (by using "new" keyword), and rest internally use a "new" keyword. All the methods dynamically allocates memory and return a reference variable to instance variable.

1. Using "new" keyword : It is the most common & general way to create an object in Java.

Eg: Test obj = new Test();

2. Using Class.forName (String className) method : There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the class object associated with the ~~name~~ with the given string name. We have to give fully qualified name for a class. On calling the newInstance() method on this Class object returns a new instance with the given string name.

Eg: // Creating object of public class Test
// Consider class Test present in com.p1 package.

Test obj = (Test) Class.forName("com.p1.Test").newInstance();

3. Using clone() method : clone() method is present in the Object class. It creates and returns a copy of the object.

Eg:
```
Test t1 = new Test(); // creating object
Test t2 = (Test)t1.clone(); // creating clone above object.
```

4. Deserialization : De-serialization is a technique of reading an object from the saved state in a file

Eg:
```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

N.B. : ⊛ By using (.) dot operator we can access instance variable.

Instance variable: Variables that are declared inside the class. but outside of the methods.

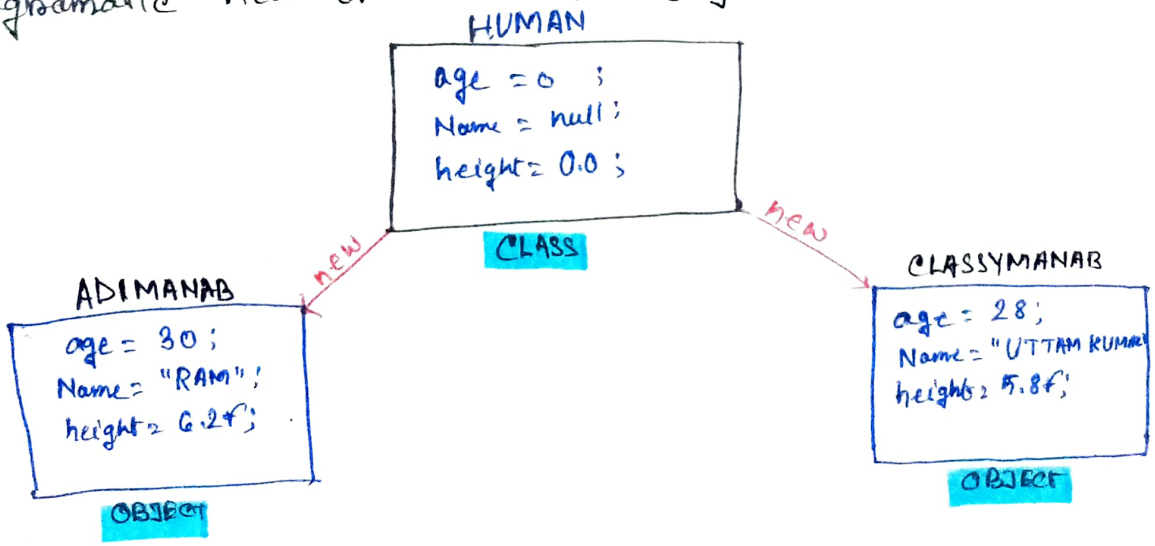
⊛ `Human obj = new Human();`
(Human obj : Compile Time; new Human(); : Run Time)

HUMan(); → It is a constructor. It is default constructor.

RunTime: After all the java code is compiled and converted into bytecode & all the checks is done then when the code is executing or running the memory is dynamically allocated.

It simply means that it will occupy memory when the program is Running

■ Diagramatic View of Class AND Object :



## Difference Between Class & Object :

| CLASS | OBJECT |
|---|---|
| i) Class is the blueprint or an object. | i) A object is instance of the class. |
| ii) No memory is allocated when class is declared. | ii) Memory is allocated as soon as an object is created. |
| iii) Class is a logical entity. | iii) An object is a physical entity. |
| iv) A class can only declared once | iv) Objects can be created many times. |
| v) An example of class can be car. | v) Objects of the class can can be BMW, Mercedes, Ferrari, etc. |

# :- Constructor In Java :-

- **Constructor:** In class-based, object-oriented programming, a constructor is a special type of method that is used to initialize objects.

The constructor is called when an object of a class/an instance of a class is created. Every time an object is created using the "new" keyword, atleast one constructor is called.

N.B.:- It is not necessary to write a constructor for a class. It is because the java compiler creates a default constructor (which is constructor with no arguments) if your class doesn't have any.

- **Syntax:** Let see the pseudo syntax:

```
class MainClass
{
....
MainClass(){
....
}
...
}
```

We can create an object of the side class using the statement

```
MainClass obj = new MainClass();
                    ↓
                Constructor
```

- **Difference Between Java Constructor & Java Methods:**
  - Constructor must have the same name the as the class within which it is defined. But it is not necessary for the method in Java.
  - Constructor do not return any type, while methods have the return type or void if doesn't return value.
  - Constructors are called only once at the time of object creation while, Methods can be called any number of times.

- **Need of Constructor in Java:** Constructor are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java using default constructor.

## ■ Types of Constructor in Java :

There are three types of constructor in Java are mentioned below :

i) Default Constructor

ii) Parameterized Constructor

iii) Copy Constructor.

### i) Default Constructor in Java :

A ~~parame~~ constructor that has no parameter is known as default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler doesnot create a default constructor. The default constructor can be changed into the parameterized constructor but parameterized constructor can't change the default constructor.

N.B. Default constructor provide the default values to the object like 0, null, etc. depending on the type.

### ii) Parameterized Constructor in Java :

A constructor that has parameter is known as parameterized constructor. If we want to initialize fields of the class with values, then use a parameterized constructor.

Eg:-
```
class ConstructorClass {
    String Name;
    int id;
    ConstructorClass(String Name, int ID){
        this.Name = Name;
        this.id = ID;
    }
```

```
ConstructorClass obj = new ConstructorClass("NIK PAL", 62);
```

N.B: • There are no "return value" statement in the constructor, but the constructor returns the current class instance.

• We can do constructor overloading as like as method overloading with different parameters.

• Once the parameterized constructor is ~~overload~~ declared then default constructor would not ~~work~~. work.

3. Copy Constructor in Java :

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to newly created object.

```
Eg: Class Greek {
        String Name;
        int id;
        Greek(String Name, int ID)
        {
          this.Name = Name;
          this.id = ID;
        }
        Greek(Greek obj)        <-- Copy Constructor.
        {
          this.Name = obj.Name;
          this.id = obj.id;
        }
    }
```

N.B: • By using the keyword final before datatype (Eg- final int a=10) you can not change its value if it is only a primitive data type.

• final keyword should use intializing and declaring only at a time cause you cant change its value.

## Static Variable in Java

Static is a keyword by which we can make any variable common to all the existing object and if we manipulate any data it will affect all the object's instance variable. It happens because static variable is used by all the objects as a common variable.

**Syntax:**

```
class ABC{
    int a;
    static String name;
}
```

**Keypoint of Static:**

- Rather than using object to use static variable, we can use class name itself to use static variable, as static variable is common for all the objects of the class.

Eg: `ABC.name = "Anik";`

ABC — Class Name

- In a non-static method we can use static variable. There is no error in that.
- Static creates a class member, not a object member. i.e, the variable is belongs to class. It save memory.

## Static Method in Java

The static keyword is used to construct method that will exist regardless or whether or not any instances of the class are generated. Any method that uses the static keyword is referenced to as a static method.

**Keypoints of static method:**

- A static method in java is a method is a part of the class rathers than an instance of that class.
- Every instance of a class access to the method.
- Only static data may be accessed by a static method. It is unable to access data that is not static.
- Static method can be accessed directly.

## Syntax or Static Method:

```
Access_modifier static return_type MethodName() {
    //Method Body
}
```

## Syntax to call a static Method:

```
className.MethodName();
```

N.B.: • The static method does not have access to the instance variable The JVM runs the static method first, followed by execution of the class instance. Because no method are accessible when the static method is used. A static method does not have access to instance variables, as a result static method can't have instance variable.

• Why the main method in Java is static?

It's because calling a static method isn't needed of the object. If it were a non-static function, JVM would first build an object before calling the main() method, resulting an extra memory allocation difficulty.

## Advantages & Disadvantages:

> Advantages: i. To access and change static variables and other non-object-based static method.

ii. Utility and assist classes frequently employ static method.

> Dis-advantages:

i. Non static data members or non static method cannot be used by static method and static method cannot call non-static method directly.

ii. In a static environment, 'this' & 'super' aren't allowed to be used.

# Static Block in Java:

- **Static Block in Java:** In Java support a special block called static block (also called static clause) that can be used for static initialization of a class.

  The code inside the block is executed only once, the first time the class is loaded into memory.

- **Calling of a Static Block:** There is no specified way as static blocks executes automatically when the class is loaded in the memory.

- **Pseudo Syntax:**

```
class SBlock {
    static
    {
        System.out.println("Inside the static block");
    }
}
```

- **Key points:**

(i) Static block can only use the static variables. It can't use the non-static variable.

(ii) Static block can also be executed before constructors.

(iii) A class can have any number of static initialization blocks, and they can appear anywhere in class body. The runtime system guarantees that static initialization blocks are called in order they appear in the source code.

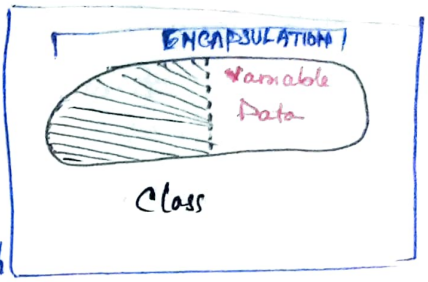
Eg.

```
class sblock {
    int i;
    static int j;
    static
    {
        System.out.println(j);
    }
}
```

# : Encapsulation In Java :

■ **Encapsulation :-** Encapsulation is defined as the wrapping up of data under a single unit through data members & methods.

Another way to visualize is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.




■ **Keypoints :-**

› Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function/method of its own class in which it is declared.

› As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private and the class is exposed to end-user or the world without providing any details behind implementation using the abstraction concept, So, it is also known as a combination of data-hiding and abstraction.

› Encapsulation can be achieved by declaring all the variables in the class as **private** and writing **public** methods in the class to set and get the values of variable.

› It is more defined with setter and getter method.

■ **Eg.:**

```
class Village{
        private String name;
        private int people;
        public void setName (String Name)
        {
          this.name = Name;
        }
        public String getName (){ return name;}
}
```

```
public class Encapsulation {
    public static void main (String[] ang) {
        Village vill1 = new Village();
        vill1.setName ("Groas");
        S.o.u.t(vill1.getName ());
    }
```

N.B: "this" keyword, refers to the current object in a method or a constructor. or The most common use of 'this' keyword is to eliminate the confusion between class and attributes with same name.

## Naming Conventions in Java

Naming Convention or having proper name gives your code more readability. Here are some types of naming conventions,

| | | |
|---|---|---|
| Class & Interface | → | Calculator, Machine |
| Variable & method | → | height, mark, myData(), setName() |
| Constants | → | PIE, CONST |
| Constructor | → | Human() |

## Anonymous Object in Java

- Anonymous Object: In java, an anonymous object is an object that is created without giving it a name. Anonymous objects are often use to create objects on the fly and ~~arguments to methods~~ pass them as arguments to methods.

Syntax: new Class_name();

# Inheritance in Java

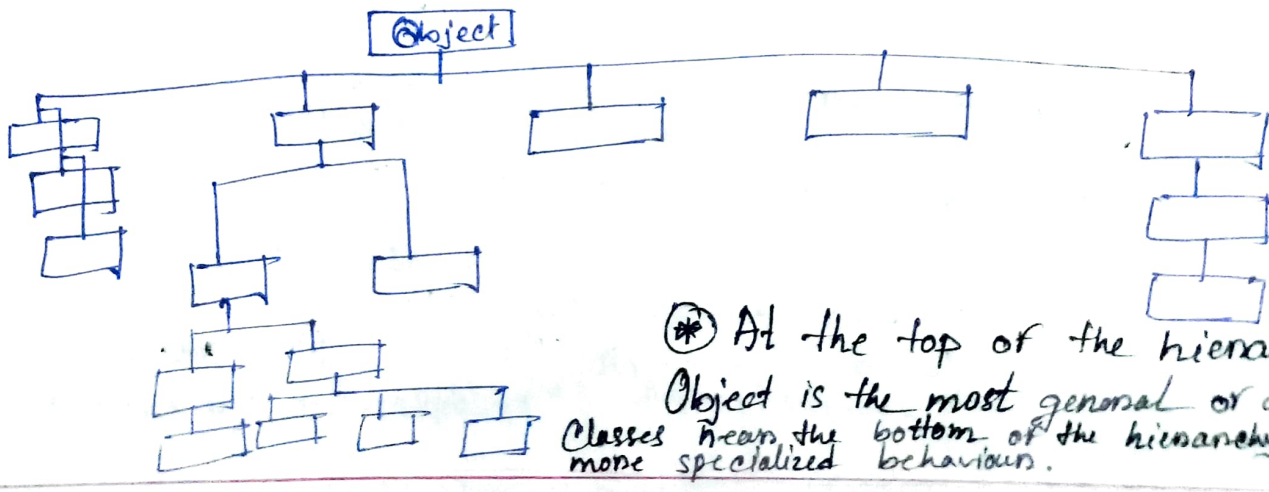
- **Inheritance :-** Inheritance is the mechnism by which one class is allowed to inherit the features (fields & methods) of another class. It simply means creating new classes based on existing ones.

When a user want to create a new class and there is already a class that includes some code that the programmer want, he/she can derive new class from existing ones.

- **Important Terms :**
  - **Subclass :** A class that is derived from another class is called a subclass. [other names 'derived class', 'extended class', or 'child class']
  - **Super Class :** The class from which a subclass is derived is called a super class. [also known as 'base class', or 'parent class'] Eg : Super()
  - **N.B :** >Except 'Object', which has no 'superclass', every class has one direct super class (single inheritence). In the absence of any other explicit superclass, every class is implicitly a sub class of 'Object'.

  > A subclass inherits all the members (eg: fields, method, nested classes) from it super class. Constructor are not members, so they are not inherited by subclass, but the constructor of the superclass can be invoked from the subclass.

- The Java Platform Class Hierarchy:




(*) At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behaviours.

## Syntax of Inheritance of a Class:

```
class sub/child/derived-class extends base/super/parent-class
{
  //method and fields
}
```

N.B.: The extends keyword is used for inheritance in java. Using the extends keyword indicates the present class is derived from existing class. In other words 'extends' refers increased functionality.

## Private Members in Super Class:

> A subclass does not inherit the **private** members of its parent class. However, if the superclass has **public** or **protected** methods for accessing, these can also be used by the subclass.

> A nested class has access to all the private members of its enclosing class - both fields & methods. Therefore, a public or protected nested class inherited by a sub class has indirect access to all of the private members of the superclass.

## Example:

```
public class BasicCalc {
    public int sum (int a, int b) { return a+b; }
    public int sub (int a, int b) { return a-b; }
}

public class AdvCalc extends BasicCalc
                [Sub Class]      [Super Class]
{
    public double div (int a, int b) { return a/b; }
    public int multi (int a, int b) { return a*b; }
}
public class Calculation {
        public static void main (String[] args) {
        AdvCalc cal = new AdvCalc();
        System.out.println ("Summation: " + cal.sum (10,15));
        System.out.println ("Multiplication: " + cal.Multi (3,5));
        }
}
```

## Java Inheritance Types:

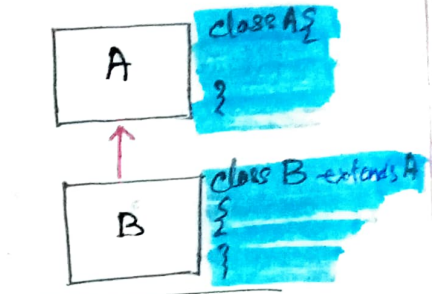
Below are the types of inheritance that are supported by java:

i. Single Inheritance
ii. Multilevel Inheritance
iii. Multiple Inheritance
iv. Hierarchical Inheritance
v. Hybrid Inheritance.

### i. Single Inheritance:

In single inheritance, subclasses inherit the features of one superclass.

Eg.: From the image beside, class A serves as a base class for the derived class B.




Single Inheritance.

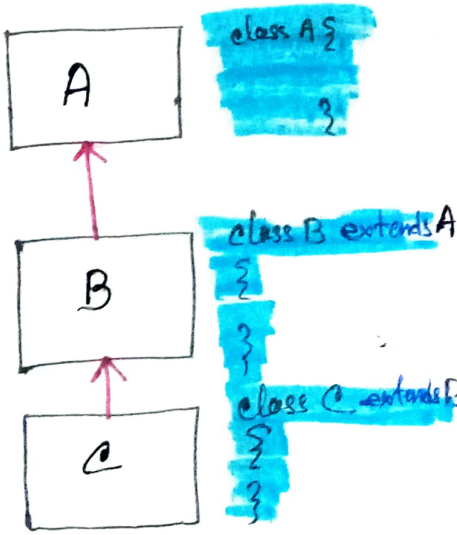
### ii. Multilevel Inheritance:

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 others classes.

Beside image gives an example.

Eg: Class A serves as a base class for the derived class B, which in term serves as a base class for derived class C.

In java, a class cannot directly access the grandparent's members.



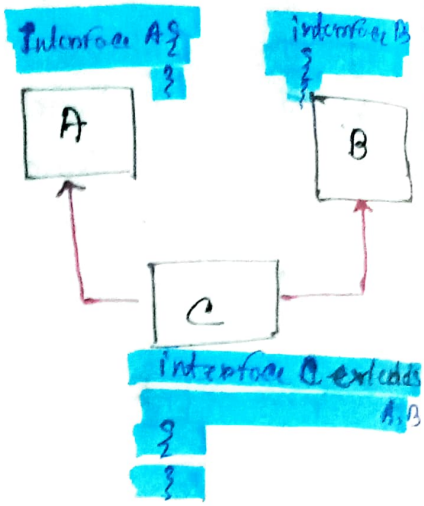

Multilevel Inheritance.

## iii. Multiple Inheritance :

In Multiple Inheritance, one class can have more than one superclass and inherit from all parent classes. Please note that **Java does not support multiple inheritance with classes.**

In java we can achieve with only through **Interfaces.**

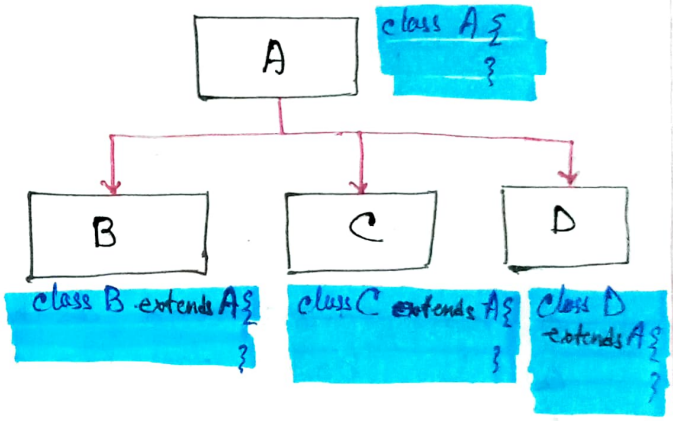
Eg: Class C is derived from interfaces A and B.




## iv. Hierarchical Inheritance :

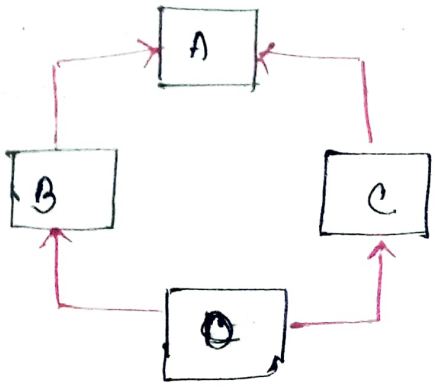
In Hierarchical Inheritance, one class serves as a superclass (Base class) for more than one subclass.

Eg: Class A serves as a base class for derived classes B, C & D.




## v. Hybrid Inheritance :

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritance with classes, **hybrid inheritance** using **multiple inheritance is also not possible** with classes. It can be achieved only through **interfaces** if we want to involve multiple inheritance to implement hybrid inheritance.



## Advantages & Disadvantages of Inheritance in Java:

Advantages:

i. Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and method of superclass, reducing duplication of code.

ii. Inheritance abstracts classes that have common interface.

iii. Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.

iv. Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms.

Disadvantages:

i. Inheritance can make the code more complex and harder to understand. It can happen when the inheritance hierarchy is deep or if multiple inheritance is used.

ii. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

## Method Overriding in Java:-

**Method Overriding:** Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided, with same name, same parameters & same return type, by one of its super/parent class.

When a method in a subclass has the same name, the same parameters or signature, and the same return type as a method in its super-class, then the method in the sub-class is said to be override the method in super-class

**Rules for Java Method Overriding:-**

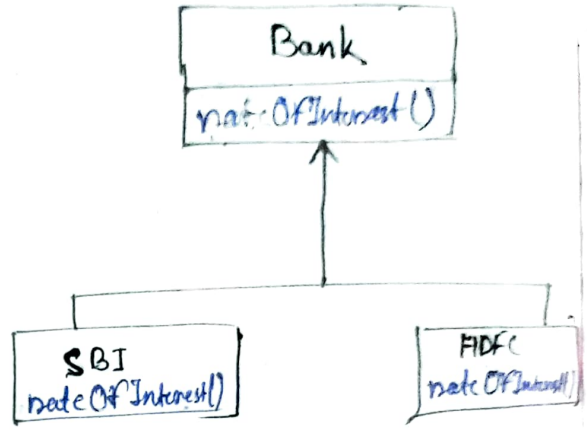
i. The method must have the same name as in the present class.
ii. The method must have the same parameter.
iii. There must be an inheritance relationship.

N.B.: > 'final', 'private', 'static method' cannot be overridden

> The reason we can't override static method, because, the static method is bound with class where as instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Example:

```
class Bank{
   float rateOfInterest(){ return 0.0f;}
}

class SBI extends Bank{
@Override float rateOfInterest(){ return 6.6f;}
}
class HDFC extends Bank{
@Override float rateOfInterest(){ return 8.2f;}
}

public class OverridingMethod(){
      public static void main (String[] args){
         SBI bankSbi = new SBI();
         System.out.println("The rate of interest in SBI is :"+
                                                  bankSbi.rateOf-
                                                  Interest());
         HDFC bankHdfc = new HDFC();
         System.out.println("The rate of interest in HDFC is :"+bankHdfc.
                                                  rateOfInterest());
      }
}
```



## Packages in Java

- **Package:** A package is grouping of classes, sub-packages & interfaces (enumerations and annotation) types.

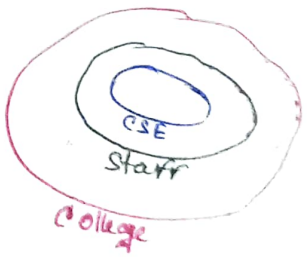
OR

Package in Java is a mechanism to encapsulate a group of classes, sub-packages and interfaces.

- **Usage of Packages:**
  - > Preventing naming conflicts.
  - > Makes searching/locating and usage of classes, interfaces, enumerations and annotation easier.
  - > Provides controlled access to the packaged elements.
  - > Packages can be considered as data encapsulation.

**N.B.:** Package names and directory structure are closely related. For example if a package name is college.staff.cse, then there are directories - college, staff & cse such that, 'cse' is present in 'staff' and 'staff' is present inside 'college'.

Also, the directory of 'college' is accessible through 'CLASSPATH' variable. i.e. path of the parent directory of 'college' is present in 'CLASSPATH'.




- **Package Naming convention:** Packages are named is reverse order of domain name, ie. com.google.spring. For example, college.tech.cse, college.arts.history.

■ Adding a class to a package :> We can add more classes to a created package by using package name at the top of the program and saving it in the package directory.

> Subpackages: Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly.

Example - import java.util.*; [util is a subpackage created inside java package]

■ Accessing classes inside a package:

> Accessing single element :→ import java.util.Scanner;

> Accessing all elements :→ import java.util.*;

■ Types of Packages :-

Package

In-Built Package

> These packages consists of a large number of classes, which are a part of Java API.

Some of the commonly used built-in packages are:

i) java.lang : Contains language support classes. (imported automatically)
ii) java.io : Contains classes for supporting input/output operation.
iii) java.util : Contains utility classes which data structures like linked list, Dictionary and support.
iv) java.applet: Contains classes for creating Applets.
v) java.awt : Contains classes for implementing the component GUI.
vi) java.net : Contains classes for supporting networking operations.

User Defined Package

> These are the packages that are defined by the user. First we create 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 name';

N.B: In package only the classes that are used are stored. The main program will be stored outside of the package.

# Access Modifiers in Java

- **Access Modifiers:** Access modifiers in Java are the keywords that are used to control the use of methods, constructors, fields and methods in a class. It helps to restrict the scope of a class, constructor, variable, method or data member.

- **Types of Access Modifiers:**
  These are four types of access modifiers available in java:
  i) Default.
  ii) Private
  iii) Protected
  iv) Public

> i) **Default Access Modifier:**
  When no access modifier is specified for a class, method or data member – it is said to be having default access modifier by default.

  The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible only within the same package.

ii) **Private Access Modifier:**
  The private access modifiers are specified using the keyword **private**. The methods or data members declared as private are accessible only within the class in which they are declared.

  - Any other class of the same package will not be able to access these members.
  - Top-level classes or interfaces can not be declared as private because, Private means only visible within the enclosing class.

iii) **Protected Access Modifier:**
  The protected access modifier is specified using the keyword **protected**. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

iv) Public Access Modifier:

The public access modifier is specified using the keyword **public**.

- The public access modifier has the widest scope among all other modifiers.
- Classes, methods or data members that are ~~accessible~~ declared as public are accessible from everywhere in the program.

| Access Modifier / Accessibility | Default | Private | Protected | Public |
|---|---|---|---|---|
| Same Class | Yes | Yes | Yes | Yes |
| Same Package (sub-class) | Yes | No | Yes | Yes |
| Same Package (Non Subclass) | Yes | No | Yes | Yes |
| Different Package ~~Non~~ (Subclass) | No | No | Yes | Yes |
| Different Package (Non-Subclass) | No | No | No | Yes |

N.B: 12 modifiers in Java are public, protected, default, private, final, synchronized, abstract, native, strictfp, transient and volatile

# Polymorphism in Java

■ **Polymorphism**:- The word polymorphism means having "many forms". Polymorphism allows to perform single action in different ways. In other words allows to define one interface and have multiple implementations.

■ **Types of Java Polymorphism**:

In Java polymorphism is mainly divided into two parts/types: (i) Compile-Time Polymorphism
(ii) Runtime Polymorphism.

(i) **Compile-Time Polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by the method overloading or operators overloading.

[N.B. Java does not support operators overloading.]

Eg:
```
class Multiplication {
    static int multiply(int a, int b){ return a*b;}
    static double multiply(double a, double b){ return a*b;}
}

Public class PolymorphismDemo{
    public static void main(String args[]){
        System.out.println(Multiplication.multiply(3.14, 2.13));
        System.out.println(Multiplication.multiply(3, 2));
    }
}
```

(ii) **Runtime Polymorphism in Java**: It is also known as Dynamic Method Dispatch. It is process in which a function call to the overridden method is resolved at runtime. Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class.

Eg: a overridden method.

N.B: Virtual Method / Functions: It allows an object of a derived class to behave as if it were an object of the base class. The derived class can override the virtual function/method of the base class to provide its own implementation. The function/method call is depending on the actual type of object.

■ Advantages of Polymorphism:

i. Increase code reusablity by allowing objects of different classes to be treated as a object of a common class.

ii. Improves readability and maintainability of the code.

iii. Support dynamic binding, enabling the correct method to be called at runtime based on the actual class of the object.

iv. Enables objects to be treated as a single type, making it easier to write generic code.

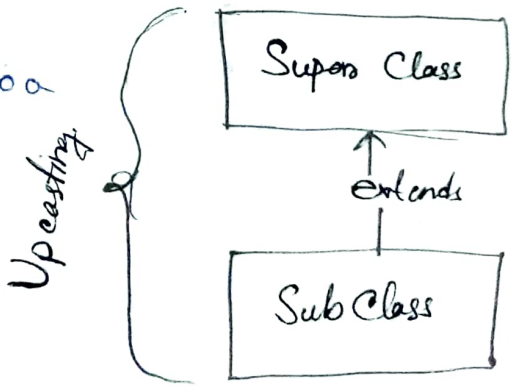
■ Disadvantages of Polymorphism:

i. It can make more difficult to understand the behaviours of an object.

ii. This may lead to performance issue, as polymorphic behaviour may require additional computation at runtime.

N.B:- Dynamic Method Dispatch: It is nothing but the Runtime polymorphism.

• A superclass reference variable can refer to a subclass object. This is also known as upcasting.




SuperClass obj = new SubClass();

Eg:
```
class Computer {
    void show() {
    System.out.println("In Computer's Show");
    }
}
class Laptop extends Computer {
    void show() {
    System.out.println("In Laptop's show");
    }
}
```
— Here, Computer obj = new Laptop(); (Valid)
Laptop obj = new Computer(); (Invalid)

# Final Keywords in Java

■ **Final Keyword:** In Java, the final keyword is used to indicate that a variable, method or class cannot be modified or extended.

> **Final Variable:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.

↳ **Syntax:** final data_type variable_name ;

↳ **Eg:** final double PI = 3.14;

OR

final double PI ;
PI = 3.14 ; ] - local final variable.

[ N.B: When a final variable is created inside a method/constructor/Block, it called local final variable, and it must initialize once it is created. where.

> **Final Classes:** When a class is declared as final, it cannot be extended by a subclass. This is useful to stop the inheritance.

↳ **Syntax/Eg:**

```
final class A
{

}
class B extends A
{
// This not possible As A is final
}
```

```
final class B extends A
{

}
final class C extends B {
// This is not possible as B is final.
}
```

> **Final Method:** When a method is declared as final, it cannot be overridden by a subclass. This is useful for stopping the overriding as well as a part of class's public API and should not be modified by subclass.

↳ **Syntax/Eg:**

```
class A {
final int add (int a, int b) {
        return a+b;}
}
class B extends A{
  int add (int a, int b){
        return a+b
  }
}
```

// Compile - Error

## Advantages of Final:

(i) Ensuring Immutability

(ii) Improving performance: The use of final can sometimes help improve performance, as JVM can optimize code more efficiently when it knows certain values or references can not be changed.

(iii) Making code easier to understand.

(iv) Promoting code reuse.

(v) Enhancing Security: The use of final keyword can help by enhancing security by preventing malicious code from modifying sensitive data or behaviours.

N.B:

Final Variable ⟶ To Create constant variable.

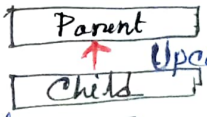
Final Method ⟶ Prevent Method Overriding

Final Class ⟶ Prevent Inheritance

## Upcasting VS Downcasting in Java

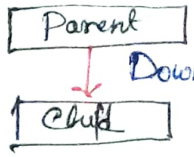
Typecasting is one of the most important concept which is basically deals with the conversion of one data type to another data type implicitly or explicitly. There are two types of typecasting,

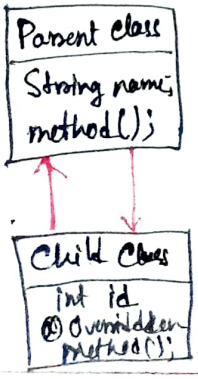
**1.) Upcasting:** Upcasting is the typecasting of a child object to a parent object. Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this features. **Syntax:** Parent p = new Child();




**(ii) Downcasting:** Similarly, downcasting means the typecasting of a parent object to a child object. Downcasting cannot be implicit.

**Syntax:** Child c = (Child) p




```
Parent P = new Child(); // Upcasting
P.name = "Pnabin Kr. Pal"; //[P.id is not accessible]
P.method()
Child c = (Child) P; //down casting.
c.id = 12.8
c.method();
// Child c = new Parent(); is not possible.
```

# Wrappers Classes in Java

- **Wrappers Class:** A Wrappers class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrappers class, it contains a field and in this field, primitive data types can be stored.

- **Need of Wrappers Classes:**
  They convert primitive data types into objects. Objects are needed if we wish to modify the arguements passed into a method.

  Advantages of using wrappers class–
  1. Collections allowed only object data
  2. Cloning process only object
  3. Object data allowed null values. etc.

- **Primitive Data Types and their Corresponding Wrappers Class:**

| Primitive Data Type | Wrappers Class |
|---|---|
| char | Character |
| byte | Byte |
| short | Short |
| int | Integers |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |

## Autoboxing and Unboxing:

**Autoboxing:** The automatic conversion of primitive types to the object of their corresponding wrapper class is known as autoboxing.

**Example:** Conversion of int to Integer, long to Long, double to Double etc.

```
int num = 12;
Integer obj = num;
S.O.U.T (obj); [obj = 12]
```

```
Integer a = 8; ← Autoboxing.
behind the scene
Integer a = Integer.valueOf(8);
                            Primitive Data Type
```

**Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.

**Example:** Conversion of Integer to int, Long to long, Double to double etc.

```
Integer obj = 12;
int value = obj.intValue();
S.O.U.T (value); [value = 12]
```

## Abstract Keyword in Java:-

**Abstract Keyword:-** In Java abstract is a non-access modifier in Java applicable for classes, and methods but not variables. It is used to achieve abstraction which is one of the pillars of OOP.

**Characteristics of Java abstract Keyword:** In Java, the abstract keyword is used to define abstract classes and methods.

- **Abstract classes cannot be instantiated:** An abstract class is a class that cannot be instantiated. Instead, it is meant to be extended by other classes, which can provide concrete implementations of its abstract method.

```
abstract class A { int var1; }
class B extends A { }
void main() {
    // A obj = new A(); // Invalid
    A obj = new B(); // Valid
}
```

- **Abstract method do not have a body:** An abstract method is a method that does not have an implementation. It is declared with a **abstract** keyword and ends with a semicolon instead of a method body. Subclasses of an abstract class must provide a concrete implementation of

all abstract method defined in the parent class.

Syntax of Abstract Method: access_modifier abstract return_type method_name();

Example: public abstract void display();

- Abstract can have both abstract and normal methods // Normal Methods
- Abstract classes can contain instance variables: Abstract classes can contain instance variables, which can be used by both the abstract class and its subclass.

  Eg: class First {
      public abstract void First(); // abstract constructors.
  }

- Abstract classes can implement interfaces: Abstract classes can implement interfaces, which defines a set of methods that must be implemented by any class that implements the interface.

N.B:

> Important rules for abstract methods:

↳ Any class that contains one or more abstract methods must also be declared abstract.

↳ The following are various illegal combinations of other modifiers for methods with respect to abstract modifier.

i. final
ii. abstract native
iii. abstract synchronize
iv. abstract static
v. abstract private
vi abstract strictfp.

// "access_modifier abstract" is allowed.

> Although abstract classes cannot be used to instantiated objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented. (Upcasting)

Example: <C:\Users\Anik Pal\OneDrive\Desktop\Java\OOPS_in_java\Wrappers (Abstractkeyword.java) class>

N.B: Abstract vs final keyword:

In Java you will never see a class or method with both final and abstract keywords at a same time.

For classes, final is used to prevent inheritance, where as abstract classes depend upon their child classes for complete implementation.

In cases of methods, final is used to prevent overriding, where as abstract methods need to be overridden is sub-classes.

# Inner Class in Java

■ **Inner Class:** In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced to sum up, some logically relatable classes as Java is purely object oriented.

■ **Advantages of Inner Class:**
- Making code more clean & readable.
- Private methods of the outer class can be accessed.
- Optimizing the code module.

■ **Types of Inner Classes:** There are basically fours types of inner class

1. Nested Inner Classes
2. Method local Inner Classes
3. Static Nested Classes
4. Anonymous Inner Classes

- Member Inner class
- Static Nested Class
- Local Inner Class
- Anonymous Inner Class

**Type I: Nested Inner Class:**

It can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected and default modifier.

Eg:
```
class Outer {
    /*private*/ int var1 = 10;
    /*private*/ float var2 = 20.2f;

        class Inner {
            public void show() { S.O.U.T("In Inner class"); }
            public void add() { S.O.U.T(var1 + var2); }
        }
}

class Main { public static void main(String[] args) {
    Outer.Inner obj = new Outer().new Inner();
    obj.add();
    obj.show();
    }
}
```

↳ Qualified Object Creation.

## Type 2: Method Local Inner Class:

Inner class can be declared with in a method of an outer class which will be illustrating in the below example.

Eg:
```
class Outer {
    void outerMethod() {
        S.O.P("Inside OuterMethod");
        class Inner {
            void innerMethod() {
                S.O.P("Inside InnerMethod");
            }
        }
        Inner obj1 = new Inner();
        obj1.innerMethod();
    }
class Main { public static void main(String[] args) {
        Outer.obj2 = new Outer();
        obj2.outerMethod();
    }
}
```

N.B: The Inner class is local to the outerMethod(). So, it can't be accessed outside the method.

N.B.:- Method Local Inner classes can't use a local variable of the outer method until that local variable is not declared as final.

## Type 3: Static Nested Classes:

Static nested classes are not technically inner classes. They are like a static member of outer class. Basically there is no need to create an object's reference variable for the class.

Eg: Outer.InnerMethod1(); /C:/Desktop/Java/OOPS-IN-JAVA/InnerClass.Java

Computer.Pendrive pendrive = new Computer.Pendrive("type-c");

## Type4: Anonymous Inner Classes:

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

- As a subclass of the specified type
- As an implementer of the specified interface.

Eg:

- As a subclass of a specified type

```
class Demo {
    void show display() { S.O.P. ("In Demo's Display"); }
}
class Main {
    public static void main (String[] args) {
        Demo obj = new Demo() {
            void show display() { S.O.P. ("In Main's Anonymous Class's Display"); }
        };
        obj.display();
    }
```

Output : In Main's Anonymous Class's Display

- N.B:- If you use 'anonymous inner class' to a 'abstract class', there is no need to inherit the abstract class to create a reference variable of the subclass.

Eg:

```
abstract class Demo {
        abstract public void show();
        abstract public void config();
}
class Main {
    public static void main (String [] args)
    {
        Demo obj = new Demo() {
            public void show() {S.O.P. ("In a Show");}
            public void config() {S.O.P. ("in a Config");}
        };
        obj. show();
        obj. config();
    }
}
```

# Interface in Java

- **Interface**: The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body/implementation. It is used to specify the behaviours of a class and multiple inheritances in Java using Interfaces. It also represents the IS-A relationship.
  - Blueprint for class. It holds static constant variable.
- **Advantages of Interface**:
  i) It is used to achieve abstraction.
  ii) By interface, we can support the functionality of multiple inheritance.
  iii) It can be used to achieve loose coupling
  iv) Any class can extend only 1 class, but any class can implement an infinite number of interface.
- **Syntax of Interface**:
  To declare an interface, use the interface keyword. It is used to provide total abstraction. The methods are public, abstract and static; and the variables are public, static & final, by default.

```
interface  interface's_name
{
  // declare constant fields.
  // declare method (public & abstract by default);
}
```

A class that implement interface must implement all the methods declare in the interface. To use interface in a class use the keyword **implements**.

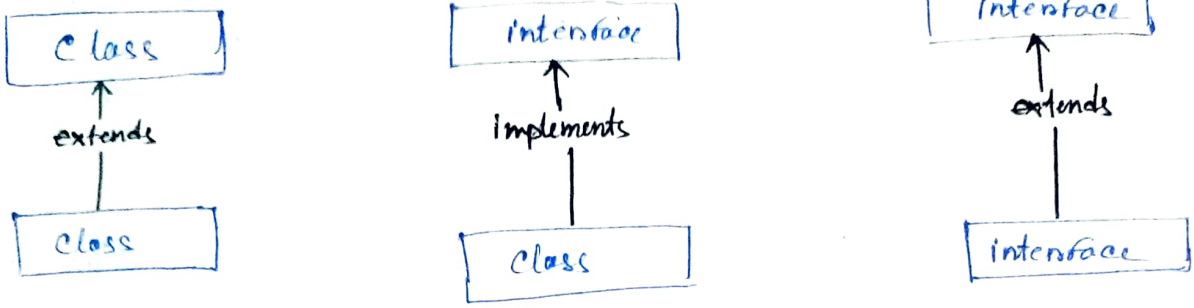
---

N.B.: Why we use interfaces when we have abstract classes?
Because, abstract classes may contain non-final variables, whereas variables in the interfaces are final, public and static.

Eg:
```
interface Human
{
   final int teeth_count = 32;
    void eat(); //[Here the method is abstract public by default]
}
```

## Relationship Between Class and Interface:

A class can extend another class similar to this an interface can extend another interface. But only a class can implement to another interface, but vice-versa is not allowed.




## Difference Between Class and Interface:

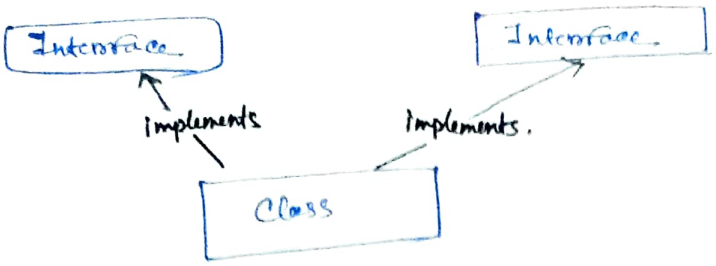
| Class | Interface |
|---|---|
| i) In class, you can intantiate reference variable and create an object. | i) In an interface, you can't intantiate variables & create an object. |
| ii) A class can contain concrete methods (except is some abstract class). | ii) The interface cannot contain concrete methods, it have only abstract method. (N.B – use static keyword you can make concrete method.) |
| iii) The access specifiers used with classes are private, protected and public. | iii) In Interface only one specifier is used - public. |

Example: ① Desktop/Java/OOPS_in_Java/Interface/Basicinheritance.java

② Desktop/Java/OOPS_in_Java/Interface/Realifeinheritance.java

## Multiple Inheritance in Java Using Interfaces:

Multiple inheritance in an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritance in Java using interfaces.




Eg:
```
interface First{
        void display();
}
interface Second{
        void show();
}
class Merge implements First, Second{

    public void display(){ S.O.P("In First Interface");}
    public void show(){ S.O.P("In Second Interface");}
  public static void main(String[] args){
    Merge obj = new Merge();
     obj.display();
     obj.show();
  }
}
```

## ■ Extending Interfaces:

One interface can inherit another interface by the use of extends keyword. When a class implements an interface that inherit another interface, it must provide an implementation for all the method required by the interface inheritance chain.

Syntax:
```
interface Parent_interface
{
  // abstract methods
}
interface Child_interface extends parent_interface
{ // abstract methods.
}
```

Eg: Desktop/Java/OOPS_in_Java/Interfaces/Extendinginterfaces.java.

## Types of Interface:

There are basically 3 types of interfaces —

i) Normal Interface
ii) Functional Interface
iii) Marker Interface

**i) Normal Interface:** When an interface contains more than one abstract method, then it is known as Normal Interface.

Example:
```
interface Human{
        void eat();
        void sleep();
}
```

**ii) Functional Interface:** When a interface contains only one abstract method, then it is known as a Functional Interface.

Example:
```
@FunctionalInterface
interface Car {
    void drive();
}

class Main {
    public static void main(String[] args)
    {
        Main obj = new Main(){
            public void drive(){ s.o.p("Driving...");}
        };
        obj.drive();
    }
}
```

/** Functional Interface using Anonymous Inner Class */

[(Desktop/JAVA/OOPS_In_Java/Interface/FunctionalInterface.java]

**iii) Marker Interface:**

- An interface that does not contain any methods, fields, Abstract Methods, and any Constants is called a Marker interface.
- Also, if an interface is empty, then it is known as Marker Interface.
- The serializable and the cloneable interfaces are the example of marker interfaces.

Example: public interface interface_name{
    // Empty.
}

There are two alternatives to the marker interface that produce the same result as the marker interface.

1) Internal Flags – It is used in the place of the Marker Interface to implement any specific operation.

2) Annotations – By applying annotations to any class, we can perform specific actions on it.

■ Built-in Maker Interface:

There are three types of Built-In Marker Interface in Java. Like,

i) Cloneable Interface → [A cloneable interface in Java is also a Marker Interface that belongs to java.lang packages.]

ii) Serializable Interface

iii) Remote Interface.

i) Cloneable Interface:

• A cloneable interface in java is also a Maker interface that belongs to java.lang packages

• It generates a replica of an object with a different name. Therefore we can implement the interface in the class of which class object is to be cloned.

• It implements the clone() method of the object class to it.

N.B.: A class that implements the Cloneable interface must override the clone() method using a public method

ii) Serialization Interface:

• It is a marker interface in Java that is defined in the java.io package. If we want to make the class serializable, we must implement the Serializable interface. If a class implements the serializable interface, we can serialize or deserialize the state of an object.

- Serialization is a mechanism in which our object state is ready from memory and written into a file or from databases.
- De serialization - is the opposite of serialization means that object state reading from a file or database and written back into memory is called de serialization of an object.

N.B.:
Serialization - Converting an Object into byte stream.
Deserialization - Converting byte stream into an object.

iii) Remote Interface :

- A remote interface is a marker interface that belongs to java.rmi package. It marks an object as a remote that can be accessed from the host of another machine.
- We need to implement the Remote interface if we want to make an object remote then. Therefore, It identifies the interface.
- A remote interface serve to identifies interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface.
- The remote interface is an interface that declares the set of methods that will be invoked from a remote Java Virtual Machine.

```
interface Animal{
    void eat();
    void sleep();
    default void run(){
        sout("Running");
    }
}
```

```
Animal.run()
```

```
class Dog implements Animal{
}
Dog dog = new Dog();
dog.run()
```

default gives the default implementation automatical without in the implemented class without any implementation.

# Enumerations or Enum in Java

■ **Enum :** An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. A Java enumeration is a class type. Although we don't need to instantiate an enum using `new`, it has the same capabilities as other classes.

Enumeration or Java Enum serve the purpose of representing a group of named constants in a programming language. Java Enums is used when we know all possible values at compile-time, such as choices on a menu, rounding modes, command-line flags, etc. The set of constants in an enum type doesn't need to stay fixed for all time.

■ **Declaration of enum in Java :**

N.B. : Enum declaration can be done outside a class, or, inside a class but not inside a method.

› **Declaration outside the class –**

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY THURSDAY , FRIDAY, SATURDAY;
}

public class Main {
    public static void main(String [] args) {
        Day d = Day.Monday;
        S.O.P("The day is: "+d);
    }
}
```

› **Declaration inside the class –**

```
public class Main{
    enum Colon {
        BLUE, RED, GREEN, YELLOW;
    }
    public static void main(String[] args){
        Colon c = Colon.BLUE;
        S.O.P("The colon is:"+c);
    }
}
```

- The first line inside the **enum** should be a list of **constants** and then other things like **methods, variables** and **constructors.**

- It is recommended that we name constant with all capital letters.

## ■ Properties of Enum in Java:

These are certain properties followed by Enum as mentioned below:

- Every enum is internally implemented by using class.
- Every enum constant represents an object of type enum.
- Enum type can be passed as an argument to **switch statements.**
- Every enum constant is always implicitly public static final. since it is **static**, we can access it by using the **enum Name.** Since, it is **final** we can't create **child enums.**
- We can declare the main() method inside the enum. Hence we can invoke directly from the command prompt.

## ⊛ Switch Statement by passing the enum constants:

```
enum Light {
    RED, GREEN, YELLOW; }        // Desktop/Java/OOPS_In_Java/Enum
                                    (TrafficSignal.jav

class Main {
    public static void main(String[] args){
        Light lt = Light.GREEN;
        switch (lt) {
            case RED:
                S.O.P("STOP");
                break;
            case GREEN:
                S.O.P("Drive"); break;
            case YELLOW:
                S.O.P("Ready!");
                break;
            default:
                S.O.P("please improve your colour Recognization");
        }
    }
}
```

* Loop through Enum:
We can iterate over the Enum using value() and loop. values() function returns an array of Enum values as constants using which we can iterate over values.

Example:
```
enum Family {
        RABA, MAA, BUNU, GIRLFRIEND;
}
class Main {
    public static void main(String[] args) {
        for(Family itr : Family.values())
        {
            s.o.p(itr);
        }
    }
}
```

* Main Function Inside Enum: We can declare a main function inside an enum as we can invoke the enum directly from the command ~~prompt~~ prompt.

Example:
```
enum Colon{
        ORANGE, WHITE, GREEN;

        public static void main(String[] args)
        {
            Colon C1 = Colon.WHITE;
            System.out.println(C1);
        }
}
```

■ Enum and Inheritance:
- All enums implicit extends java.lang.Enum class. As a class can only extend one parent one parent in java, so, enum cannot extend anything else.
- toString() method is overridden in java.lang.Enum class, which returns enum constant name.
- enum can implement many interfaces.

## ■ Enum and Constructor :

- Enum can contain a constructor and it is executed seperately for each enum constant at the time of enum is Class loading.
- We can't create enum objects explicitly and hence we can't invoke the enum constructor directly.

## ■ Enum and Methods :

- Enum can contain both concrete methods and abstract methods.
- If an enum class has an abstract method then each instance of the enum class must implement it.

Example: Desktop/Java/OOPs_In_Java/Enum/(EnumAbstractMethod.java)

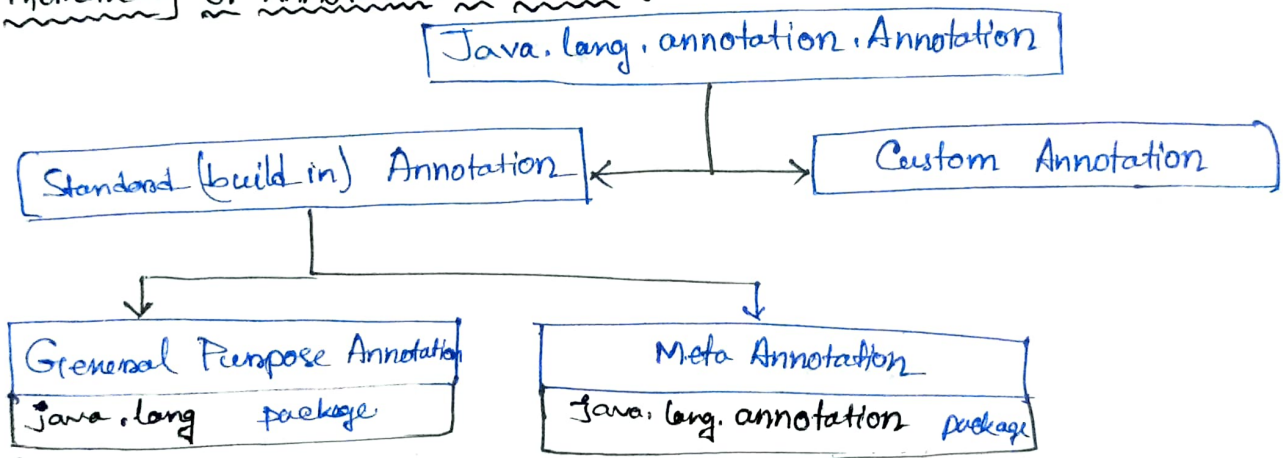
```
Example: enum Country{
            INDIA {
              @Override
              public void economy(){ S.O.P("This.toString() +" has 3nd largest Economy");
                                   }

            }
            ,AUSTRALIA{
              @Override
              public void economy(){ S.O.P(this.toString() +"has 5th largest Economy");
                                   }

            }
          public abstract void economy(); /* abstract enum method */
          }
```

# Annotations in Java

■ **Annotations in Java:** Annotations are used to provide supplemental information about a program.

- Annotations start with '@'.
- Annotations do not change the action of a compiled program.
- Annotation helps to associate metadata to the program elements. (i.e. instance variable, constructor, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by the compiler.
- Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interface.

■ **Hierarchy of Annotation in Java:**




Eg. @Override
@Deprecated
@SafeVarArgs
@SuppressWarnings
@FunctionalInterface

Eg @Inherited
@Documented
@Target
@Retention
@Repeatable.

# Lambda Expression in Java :-

- **Lambda Expression :** In Java, Lambda expressions basically express instance of functional interfaces. Lamda Expression in Java are basically same as lambda functions which are the short block of code that accepts input as parameter and return value as resultant value.

- **Functionality of Lambda Expression :**

  Lambda Expression implement the only abstract function, therefore implement functional interfaces lambda expressions are added in Java 8.

  - Enable to treat functionality as a method argument, or code as data
  - A function that can be created without belonging to any class
  - A lambda expression can be passed around if it was an object and executed on demand.

- **Lambda Expression Syntax :**

  argument_list -> body_of_lambda_expression. ;

like ; (int arg1, String arg2) -> { S.O.P.(" The arguments are " + arg1 + arg2); }

Example :

```
@FunctionalInterface
interface Demo{
    void display();
}

public class Main{
    public static void main(String[] args){
        /*Demo obj = new Demo(){ }; */ // Instead use lambda expression
        Demo obj = () -> { S.O.P("Lambda Expression"); }
        obj.display();
    }
}
```

## ■ Lambda Expression Syntax:

There are three Lambda Expression Parameters are mentioned below;

i) Zero Parameter. Eg: `() -> {System.out.println("Zero parameter lambda");};`

ii) Single Parameter. Eg: `(p) -> {System.out.println("One parameter : "+p);};`

iii) Multiple Parameters. `(P1, P2) -> {System.out.println("Multiple Parameter : "+P1+","+P2);};`

---

N.B:

- It is not mandatory to use parentheses if the type of that variable can be inferred from the context.
- Lambda expression can only be used to implement Functional interfaces
- In Lambda expression if there is a only one statement as 'return' then there is no need to mention 'return'. If you don't use {}.

- If you use {}, then you have to mention ';' after the braces has been closed.

---

Example:

```
public Main{
    interface Add{
        int add(int x, int y);
    }
    interface Display{
        void display();
    }
    public static void main(String[] args)
    {
        Add a = (x,y) -> x+y;
        Display d = () -> {S.O.P("Java is Great");};
        System.out.println(a.add(10,20));
        System.out.println(d.display());
        d.display();
    }
}
```

# Exceptions in Java :-

- **Exceptions :** Exception is an unwanted or unexpected event, which occurs during the execution of a program, that disrupt the normal flow of the program's instructions. It is an object which is thrown at runtime.

Exceptions can be caught and handled by the program. When an exception occurs within a method, it create an object, which is called exception object. It contains information about the exception, such as the name and description of the exception and state of the program when the exception occurred.

- **Major reasons why an exception occurs :**
  - Invalid user input
  - Device failure
  - Loss of network connection
  - physical limitation.
  - Logical Error
  - Opening an Unavailable File.

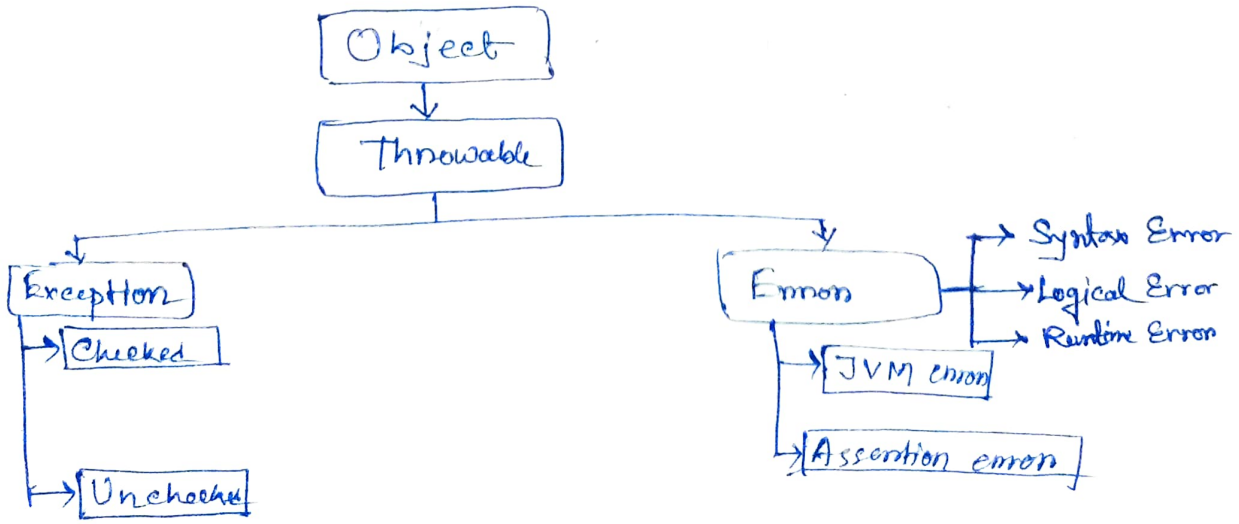
- **Errors & Errors VS Exception :**

Errors represents irrecoverable conditions such as JVM running out of memory, memory leaks, stack overflow error library incompatibilities, infinite recursion etc.

Error – An error indicates a serious problem that a reasonable application should not try to catch.

Exception – Exception indicates conditions that a reasonable application might try to catch.

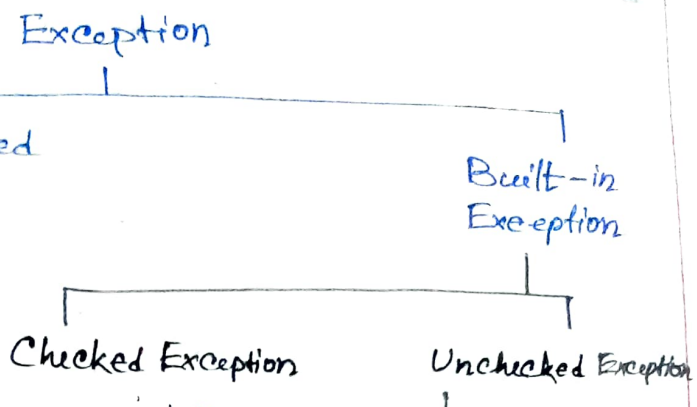
- **Exception Hierarchy :**



## ■ Types of Exceptions:

Exception can be categorized in two ways:

1. Built-in Exception
   - ↳ Checked Exception
   - ↳ Unchecked Exception
2. User defined Exception.




### 1. Built-in Exceptions:

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

↳ **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

↳ **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check this exception at compile time. In simple words, if a program throws an unchecked exception and even if we didn't handle or declare it, the program would not give a compilation error.

### 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'.

[Hierarchy of Exceptions at 106 page].

■ **Exception Handling:** Exception Handling in Java is one of the effective means to handle runtime-error so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime exception/error.

## Exception Handling By JVM:

Whatever inside a method, if an exception has occurred, the method creates an 'object' known as an Exception Object and hands it off to the runtime system (JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred.

- **Throwing:** Creating the Exception Object and handling it in the run-time system is called throwing an Exception.
- **Call Stack:** There might be a list of the methods that had been called to get to the method where an exception occured. This ordered list of method is called Call Stack.

## Block and Keywords Used For Exception Handling:

i) **try in java:** The try block contains a set of statements where an exception can occur. If a statement in try block raised an exception, then the rest of the try block doesn't execute and control passes to the corresponding catch block.

Syntax:
```
try
{
// statement that might cause exception.
}
```

ii) **catch in java:** The catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception ~~exception followed by a~~

Syntax:
```
catch
{
// Statements that handles exception
}
```

iii) **throw in java:** The throw keyword is used to transfer control from the try block to the catch block. The throw keyword is mainly used to throw custom exception.

■ Syntax:
```
try {
    throw new Exception_Name( );
}
```
Example: Desktop/Java/Basics_or_Java/CustomException.java.

iv) **throws in Java**: The **throws** keyword is used for execution handling without try & catch block. It specifies the exceptions that a method can throw to caller and does not handle itself

Syntax:
```
access_modifier returntype method_name() throws Exception_name{
    //code where exception may arise.
}
```
Example: Desktop/Java/Basic_or_Java/DuckingException.java

v) **finally in Java**: It is executed after a catch block. We use it to put some common code (to be executed irrespective of whether an exception has occurred or not) when there are multiple catch block.

Example:
```
class Division{
    public static void main(String[] args)
    {
      int a=10, b=5, c=6, result;
      try{
      result = a/(b-c);
      System.out.println("result :"+result);
      }
      catch(ArithmeticException e){
        System.out.println("Exception caught: Divided by 0");
      }
      finally{
          System.out.println("It is the final block");
    }}
}
```

# Threads in Java

■ **Thread**: A thread is a thread of execution in a program. The JVM allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not be marked as a daemon. When code running in some thread creates a new **Thread** object, the new thread has its priority initially set equal to the priority of the creating thread, and is daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which is typically calls the method named **main** of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

↳ The **exit** method of class **Runtime** has been called and the security manager has permitted the exit operation to take place.

↳ All threads that are not daemon threads have died, either by returning from the call to the **run** method or by throwing an exception that propagates beyond the run.

■ **The Concept of Multitasking:**

To help users Operating System accomodates users the privilege of multitasking, where users can perform multiple actions simultaneously on the machine. This multi tasking can be enabled into two ways:

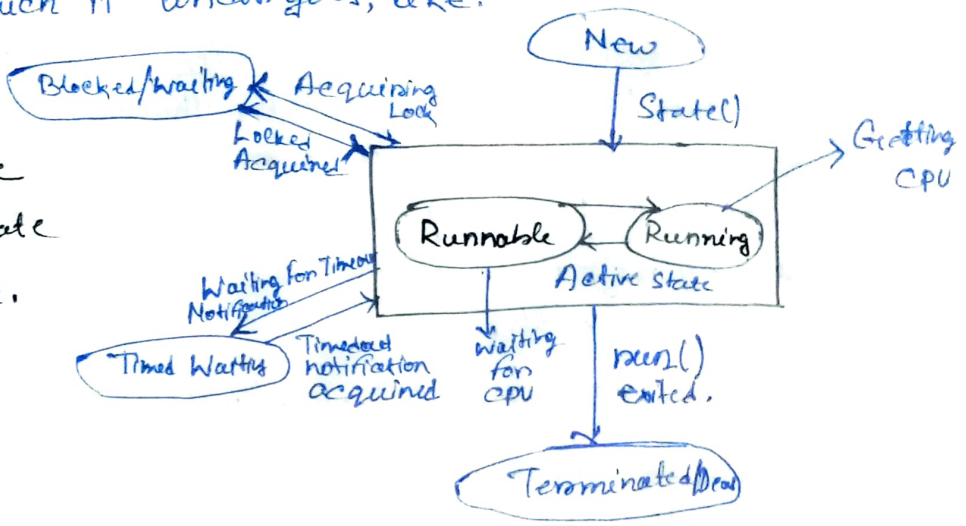
1) **Process-Based Multitasking**: In this type of Multitasking, processes are heavyweight and each process was allocated by a seperated memory area. And as the process is heavyweight the cost of communication between processes is high and it takes a long time for switching between processes as it involves actions such as loading, saving in registers, updating maps, lists etc.

ii) Thread-Based Multitasking: As we discussed above Threads are provided with lightweight nature and shares the same address space, and the cost of communication is also low.

## ■ Life Cycle Of Thread:

These are different states Thread transfers into during its lifetime, let us know about those states in the lifetime of a thread through which it undergoes, like.

i) New State
ii) Active State
iii) Waiting/Blocked State
iv) Timed Waiting State
v) Terminated state.




■ Example: Desktop/Java/OOPs_in_java/Thread/BasicThread.java.

### i) New State:-

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is yet not initiated.

### ii) Active State:-

A Thread that is a new state by default gets transferred to Active State when it invokes the start() method, his Active state contains two sub-states namely:

▶ **Runnable State:** In this state, the Thread is ready to run at any given time and its the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time interval which are shared between threads hence, these threads run for some short span of time and wait in runnable state to get their scheduled slices of a time interval.

▶ **Running State:** When the Thread receives CPU allocated by Thread Scheduler, it transfers from "Runnable" State to the "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

iii) Waiting / Blocked State:

If a Thread is inactive but on a temporary time, then either it is waiting or blocked state.

Eg: a) If there are two threads, $T_1$ & $T_2$ where $T_1$ needs to communicate to the camera and the other thread $T_2$ is already using the camera to scan, then $T_1$ waits until $T_2$ thread completes its work, at this state $T_1$ is parked in waiting for the state.

b) If both the threads having same functionality and both had same time slice given by thread scheduler the both threads $T_1$ & $T_2$ is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler cleans Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

iv) Timed Waiting State: Some-times the longer duration of waiting for threads causes, starvation, if we take an example like there are two threads $T_1$, $T_2$ waiting for CPU and $T_1$ is undergoing a Critical coding operation and if it does not exist the CPU until its operation gets executed then $T_2$ will be exposed to longer waiting with undetermined certainty. In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period which sleep() method is invoked and after the time expires the Threads starts executing task.

v) Terminated State:- A thread will be in Terminated state, due to the below reasons:

- Termination is archieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exception etc and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longers available.

■ Main Thread:

As we are familiar, we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM, Similarly in this Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM, hence whenever a program is being created in java, JVM provides the Main Thread for its Execution.

■ Thread Creation:

Threads in Java can be created in two ways—

i) Extending Thread Class

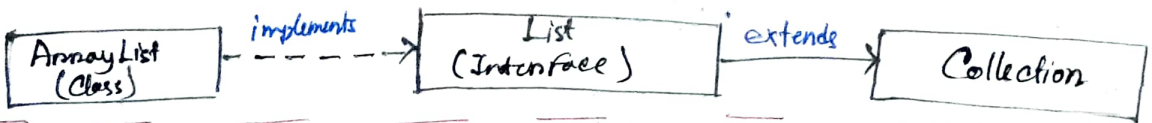
ii) Implementing Runnable interface.

## ArrayList in Java:-

■ ArrayList: Java ArrayList is a part of the Java collection framework and it is a class of java.util package. It provides us with dynamic array in Java. Though, it may be slower than standard array but can be helpful in programs where lots of manipulation is required.

The main advantages of Java ArrayList are, if we declare an array then it's needed to mention the ~~size~~ size but not in Arraylist, it is not needed to mention the size of ArrayList, If you want to mention the size then you can do it.

ArrayList is a Java class implemented using List interface.




N.B.:- Collection API → Collection (Interface)
→ Collections (Class)

■ Syntax:

~~ArrayList obj = new ArrayList();~~

→ ArrayList<Integer> = new ArrayList<Integer>(); OR

→ List<Integer> = new ArrayList<Integer>();

■ Example: Desktop/Java/Collection/Arraylistbasic.java

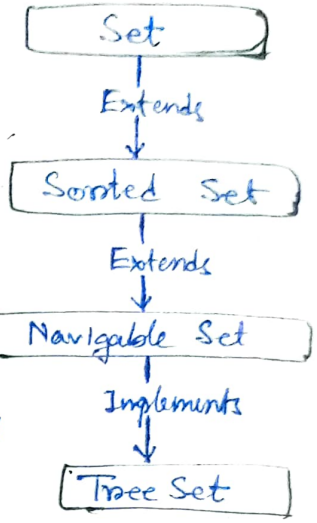
# Set in Java

- **Set:** The set interface is present in **java.util** package and extends the **Collection** interface. It is unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements mathematical set. There are two interfaces that extend the set implementation namely **SortedSet** and **NavigableSet**.

- **Creating Set Object:**
  Since set is an interface, it can't be created of the type set. We always need a class that extends this list in order to create an object. So, the obj is the type of the object to be stored in Set as given below example:

```
Set <Obj> sobj = new HashSet <Obj> ();
```

obj = Integer, Character or any other Wrapper object class or any class



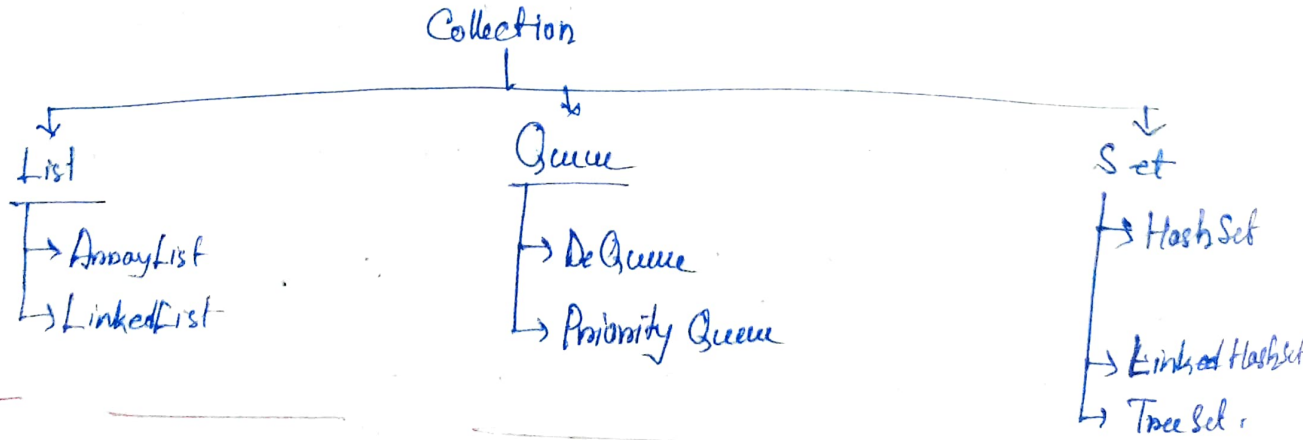

OR

```
Set <obj> sobj = new TreeSet<obj>();
```

N.B: ▸ new TreeSet<obj>(); → Created a set of sorted set.
▸ new HashSet<obj>(); → Creates a set of unsorted set.

Ex: Desktop/Java/OOPs_in_Java/Collections/Setbasic.java.

N.B.



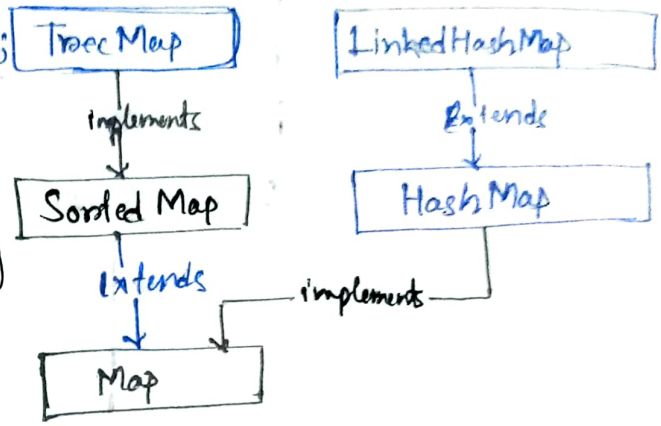
# : Map Interface in Java :

■ **Map :** In Java, Map Interface is present in java.util package represents a mapping between a key and a value. Map interface is not a subtype of the Collection interface.

Maps are perfect to use for key-value association mapping such as dictionary. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys

■ **Creating Map Objects :**

```
Map<obj1,obj2> mobj = new HashMap<obj1,obj2>();
```




■ **Example** : Desktop/Java/OOPs_in_Java/Collection/(MapBasic.java)

## Hierarchy :

- java.lang.Object
  - java.lang.Throwable
    - java.lang.Error
      - java.lang.VirtualMachineError
        - java.lang.OutOfMemoryError
        - java.lang.StackOverflowError
      - java.lang.LinkageError
        - java.lang.ClassFormatError
        - java.lang.NoClassDefFoundError
    - java.lang.Exception
      - java.lang.RuntimeException
        - java.lang.ArithmeticException
        - java.lang.NullPointerException
        - java.lang.IndexOutOfBoundsException
        - java.lang.IllegalArgumentException
      - java.io.IOException
        - java.io.FileNotFoundException
        - java.io.EOFException
      - java.sql.SQLException
      - java.lang.reflect.InvocationTargetException
      - java.net.SocketException
      - java.net.UnknownHostException

# Java Multi-threading

## Basics:

- **CPU:** CPU (Central Processing Unit) often referred to as the brain of the computer, is responsible for executing the instructions from the programs. It performs the basic arithmetic, Logic, control and input/output operations specified by the instructions.

  Eg: A modern CPU like intel core i5 or AMD Ryzen 5.

- **Core:** A core is an individual processing unit within a CPU. Modern CPUs can have multiple cores allowing them to perform multiple tasks simultaneously.

- **Program:** A program is a set of instructions written in a programming language that tells the computer how to perform a specific task.

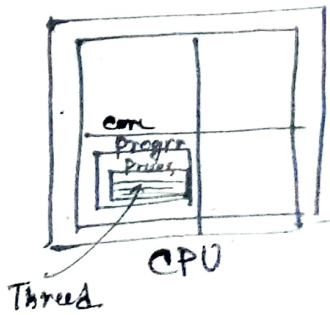
  Eg: Microsoft word is a program.

- **Process:** A process is a instance of a program that is being executed. When a program runs, the OS creates a process to manage its execution.

  Eg: When we open MS word it becomes a process in OS.

- **Thread:** A thread is a smallest unit of execution with in a process. A process can have multiple threads, which share the same resources but can run independently.

  Eg: A web browsers like Google Chrome might use multiple threads for different tabs, with each tab running as a seperate thread.



- **Multitasking:** Multitasking allows an OS to run multiple processes simultaneously. On a single-core CPU, this is done through time-sharing, rapidly switching between tasks. On multicore CPU, true parallel execution occurs, with tasks distributed accross the cores. The OS scheduler balances the load, ensuring the efficient and responsive system performance.

Multitasking utilizes the capabilites of a CPU and its cores. When an OS performs multitasking, it can assign different tasks to different cores. This is more efficient than assigning all the tasks to a single core.

- **Multithreading:-** Multithreading refers to the ability to execute multiple threads within a single process concurrently.

  - A web browser can use multithreading by having seperate threads for rendering page, running Javascript, and manages user inputs.
  - Multithreading enhances the efficieney of multitasking by breaking down individual tasks into smaller sub-tasks or threads. These threads can be processed simultaneously making better use of the CPU's capabilities.

| In a Single-Core System | In a Multi-Core System |
| --- | --- |
| Both threads and processes are managed by the OS scheduler through time-slicing and context switching to create the illusion of simultaneous execution | Both the threads and processes can run in true parallel on different cores, with the OS scheduler distributing the tasks across the cores to optimize the performance. |

## ■ Time Slicing:

- Defination: Time Slicing divides CPU time into small intervals called time slices or quanta.
- Function: The OS scheduler allocates these time slices to different processes and threads, ensuring each gets fair share of CPU time.
  - This prevents any single process or thread from monopolizing the CPU, improving responsiveness & enabling concurrent execution.

## ■ Context Switching:

- Defination: Context switching is the process of saving the state of a currently running process or thread and loading the the state of the next one to be executed.
- Function: When a process or thread's time slice expires, the OS scheduler performs a context switch to move the CPU's focus to another process or thread.

  This allows multiple processes and threads to share the CPU giving the appearance of simultaneous execution on a single-core CPU or improving the parallelism on multi-core CPUs.

## ■ Multitasking vs Multithreading:

| Multitasking | Multithreading |
|---|---|
| (i) Multitasking can be achieved through multithreading where each tasks is divided into threads that are managed concurrently. | (i) While multitasking typically refers to the running of multiple applications, Multithreading is more granular, dealing with multiple threads with in the same application or process. |

## Multitasking

(ii) Multitasking operates at the level of processes, which are the OS's primary unit of execution.

(iii) Multitasking involves managing resources between completely seperate programs, which may have seperate memory spaces and system resources.

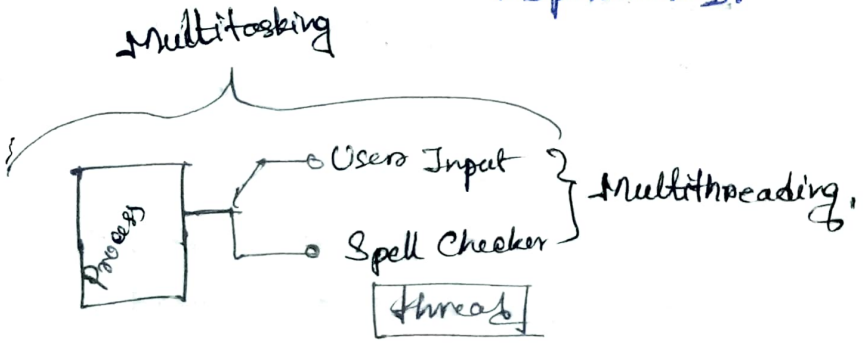
(iv) Multitasking allows us to run multiple applications simultaneously improving productivity & system utilization.

## Multithreading

(ii) Multithreading operates at the level of threads, which are smaller units with in a process.

(iii) Multithreading involves managing resources within a single program where threads shares the same memory and resources.

(iv) Multithreading allows a single application to perform multiple task at the same time, improving application performance and responsiveness.




(*) In Java, multithreading is the concurrent execution of two or more ~~process~~ threads to maximize the utilization of CPU. Java's multithreading capabilities are the part of "java.lang.package" making it easy to implement the concurrent execution.

> In a single-core environment, java's multithreading is managed by the JVM & the OS, which switch between threads to give the illusion of concurrency.

> The thread share the single-core, and time slicing is used to manage thread execution.

> In a multi-core environment, Java's multithreading can take full advantage of the available cores by JVM

- Java supports multithreading through its 'java.lang.Thread' class and the 'java.lang.Runnable' interface;

- When a java program starts, one thread begins running immediately, which is called the main thread. This thread is responsible for executing the main method of a program

- To create a new Thread in Java, one can extend the Thread Class or Implement Runnable Interface.

```
public class Hello{
    public static void main(String[] args){
        World w = new World();
        w.start();

        for(;;){
            sout("Hello");
        }
    }
}
```

```
public class World extends Thread{
    @Override
    public void run(){
        for(;;){
            sout("World");
        }
    }
}
```

```
public class World implements Runnable{
    public static void main(String[] args){
        World w = new World();
        Thread t = new Thread(w);
        t.start();
        :
    }
}
```

- A new class World is created that extends Thread.
- The run method is overridden to define the code that constitutes the new thread.
- start method is called to initiate the new thread.

■ Thread Lifecycle: The lifecycle of a thread in java consists of several states, which a thread can move through during its execution

- New: A thread is in this state when it is created but not yet started.

  Eg: World w = new World();

- **Runnable**: After the start method is called the Thread becomes runnable. It's ready to run but waiting for CPU time.
- **Running**: The thread is in this state when it is executing.
- **Blocked/Waiting**: A thread is in this state when it is waiting for a resource or for another thread to perform an action.
- **Terminated**: A thread is in this state when it has finished its execution.

Note: Usually you can use both the thread creation method but whenever a class extends another class you can use 'Thread' class over there. You have to use the 'Runnable' interface.

class A extend B, ~~extend~~ Thread{ ✗

}

class A extend B implements Runnable{ ✓

}

## Thread Methods:-

- run() - to define the operation that can be done in the thread.
- start() - creates a new thread and the run() method is executed on the newly created thread.
- join() - It is used to pause the execution of the current thread until the specified thread completes its execution.
- setPriority():- This method in Java is used to set the priority of a thread.

Syntax: setPriority(int priority)

setPriority(Thread.MAX_PRIORITY).
setPriority(Thread.MIN_PRIORITY).

- interrupt(): This method in Java is used to signal a thread that it should stop what it is doing.

- Yield(): This method in Java is a static method of the Thread class that is used to hint the thread scheduler that the current thread is willing to yield its current use of a processor. This allows other threads of the same or higher priority to be scheduled for execution.

Daemon Thread:- A daemon thread in Java is a low-priority thread that performs background operations such as garbage collection, Finalizer, Action Listeners, Signal dispatchers etc. It acts as a service provider thread that provide service to users thread.

- setDaemon(): To set a daemon thread in Java, you can use the setDaemon() method of the Thread class, setting the parameter true makes the thread a daemon, while false makes it a non-daemon thread. By default all threads are non-daemon thread.

## Thread Synchronization:

Counter.java

```
class Counter {
    private int counter;
    public int getCount(){
        return counter;
    }
    public void increment(){
        count++;
    }
}
```

CounterThread.java

```
CounterThread extends Thread{
    private Counter counter;
    public CounterThread(Counter cnt){
        this.counter = cnt;
    }
    @Override
    public void run(){
        for(int i=0; i<=1000; i++){
            counter.increment();
        }
    }
}
```

Test.java

```
class Test{
    main(){
        Counter c = new Counter();
        CounterThread t1 = new CounterThread(c);
        CounterThread t2 = new CounterThread(c);
        t1.start()
        t2.start()
        try{ t1.join()
             t2.join()
        }catch(Exception e){
        }t1.join() t2.join
    } sout(c.getCount())
```

- In the Test.java

```
Counter c = new Counter();
CounterThread t1 = new CounterThread(c);
CounterThread t2 = new CounterThread(c);
```

—— As c is a shared and common resource for both the thread t1 & t2. So the final output or value of the counter may be different & unpredictable.

(*) Critical Section: In this scenario whenever or where the shared resource get manipulated or accessed is known as Critical Section.

Here in, Counter.java

```
public void increment(){
    counter++;
}
```

CounterThread.java

```
public void run(){
    for(int i=0; i<1000; i++){
        counter.increment();
    }
}
```

Here the output of the counter field is dependent on the relative timing of both the thread, which give unpredictable output. This is known as Race Condition.

To prevent this Race Condition we use 'synchronized' keyword. So that one thread can access the Critical section at a time, this is also known as 'Mutual Exclusion'.

Mutual Exclusion ensures that the multiple section cannot access the critical section simultaneously.

Now this mutual exclusion can be achived by —

Counter.java

```
public synchronized void increment(){
    counter++;
}
```

or

```
public void increment(){
    synchronized(this){
        counter++;
    }
}
```

CounterThread.java

```
Public synchronized void run(){
    for(int i=0; i<1000; i++){
        counter.increment();
    }
}
```

■ Java Thread Locking: In Java Locking mechanisms are essential for managing access to shared resources in a multi-threaded environment, ensuring thread safety and preventing race conditions.

There are two types of locking - (i) Intrinsic
(ii) Explicit

(i) Intrinsic - These are built into every object in Java. One don't see them but they are there. When you use synchronized keyword. You are using these automatic locks.

(ii) Explicit Locks - These are more advanced locks you can control yourself using the Lock class from java.util.concurrent.locks. You explicitly say when to lock & unlock, giving you more control over how & when people/thread can use the shared resources.

▶ ReentrantLock:

ReentrantLock in Java is a part of java.util.concurrent package that helps to achieve synchronization more effectively and optimally compared to the traditional synchronized keyword. It offers features like -
- Timeout
- Interruptable Locks
- More control over Thread scheduling.

These features make it a valuable tool for managing concurrent access shared resources with greater precision and adaptability.

A ReentrantLock allows a thread to acquire the same lock multiple times, which is particularly useful when a thread needs to access a shared resource repeatedly within its execution. It implements the Lock interface, providing greater control over locking compared to synchronized blocks.

↳
- ReentrantLock tracks a "hold count", which:
- Starts at 1, when a thread first locks the resource. Each time thread re-enters the lock, the count is incremented.
- The count is decremented when the clock is released.

- When the hold count is 0, the lock is fully released.

ReentrantLock() methods :

| Methods | Description |
|---|---|
| lock() | Increment the hold count by 1 and gives the lock to the thread if the shared resources is initially free. |
| unlock() | Decrements the hold count by 1. When this count reaches zero, the resources is released. |
| tryLock() | If no other thread holds the resource, it returns true and increment the hold count. If busy, it returns false without blocking the thread. |
| tryLock(long timeout, TimeUnit unit). | The thread waits for a certain period as defined by arguments of the methods to acquire the lock on the resource before exiting. |
| lockInterruptibly() | Make Thread wait for a resource lock for a set time, interrupting if it times out or if the thread is interrupted. |
| getHoldCount() | Returns the count of the number of locks held on the resource. |
| isHeldByCurrentThread() | Return true if the lock on the resources is held by the current thread. |
| hasQueuedThreads() | Checks if there are any threads waiting to acquire the lock. |

| | |
|---|---|
| Islocked() | Queries if this lock is held by any thread. |
| newCondition() | Returns a Condition instance for use with this lock instance. |

- **Fairness of Locks :-**

```
private final Lock lock = new ReentrantLock(true);
```

→ The fairness of locks in Java, particularly ReentrantLock, refers to the mechanism that determines the order in which threads acquires a lock.

Note : Disadvantages of ~~Lock~~ Sychronized -

1. Fairness of Lock
2. Custom blocking
3. Interruptibility
4. Read/Write lock unavailable.

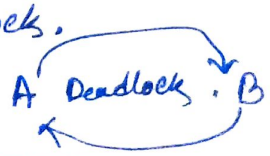
- **ReadWriteLock** - ReadWriteLock allows multiple ~~resources~~ threads to read resources concurrently as long as no threads is writing. It ensures exclusive access to write operation.

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();
Lock readLock = rwLock.readLock();
Lock writeLock = rwLock.writeLock();
```

Now one can use this locks as per their needs.

—x—

Deadlock is a situation in multithreading where two or more threads are blocked forever, waiting for eachother to release a resource. This typically occurs when two or more threads have circular dependencies on a set of Locks.



- Deadlocks typically happens when 4 conditions are met simultaneously:
  (i) Mutual Exclusion: Only one thread can access a resource at a time.
  (ii) Hold & Wait: A thread holding at least one resource is waiting to acquire addition resources held by other threads.
  (iii) No Preemption: Resources cannot be forcibly taken from threads holding them.
  (iv) Circular wait: A set of threads is waiting for each other in a circular chain.