⇒ Concepts of OOP's :-

① Classes :-

- ✱ It is a blue print to instantiate many things
- ✱ It is a "User-defined data type / ADT"
- ✱ It is a templat consisting of object or collection of object of similar type.
- ✱ The memory is not allocated when a class is created, it is allocation when object is created
- ✱ Class is a collection of data memenabers (stats / attributes) and set of methods (behaviour)

② Object :-

- ✱ It is an "Instance of a class"
- ✱ It is a basic building blocks / runtime entities that have two characteristics : states and behaviours
- ✱ Object is collection of data members and associated member functions called Methods.

③ Encapsulation :-

⊛ The Mechanism of binding code and data together into a Class is called Encapsulation

⊛ It keeps both data and code safe from outer interface and misuse.

④ Polymorphism :-

⊛ "The ability to take more than one form."

⊛ Due to this one interface can be used for General class of actions.

⊛ An object can have different meaning depending upon context.

⑤ Inheritence :-

⊛ "The Idea of Reusability"

⊛ The process by which one object acquires properties of another ⓘ the process of forming a new class from a pre-existing class ⓝ base class

⊛ It helps in reducing overall code size of Program

⑥ Dynamic Binding / Late binding :-

* It is the process of "linking the function call to the actual code of the function during runtime."

* In this, the actual code to be executed is not known to ~~Cofi~~ Compiler until run-time

⑦ Message Communication :-

* A message of an object is a "request for execution of a method / procedure."

* Objects communicate with one other by sending and recieving messages.

⇒ Object Oriented Programming :-

The Type of programming in which emphasis is on data, Data cannot be accessed by external functions as data is hidden.

⇒ Object Oriented V/s Procedure Oriented :-

Procedure Oriented	Object Oriented
Process Centric approach	Data Centric approach
Program is written around code acting on data	Program is written around Data controlling accessing the code
Programs divided into smaller ~~functions~~ parts called functions/modules	Programs divided into object which may communicate with each other using methods.
Function can call one from another and difficult to seperate due to interdependency of between modules	Objects are independent and used for different Programs
Follows Top down approach	Follows Bottoms-up approach
Ex:- C, Fortran, Pascal	Ex:- Java, C++, Smalltalk.

⇒ Advantages of OOP :-

① Simplicity → Program Structure is very clear

② Modularity → Each object form a seperate entity whose internal workings are decoupled from other parts

③ Modifiability → Easy to make minor changes in data @ procedure Changes inside a class do not effect other part of Program

④ Extensibility → Adding new features can be solved by introducing a new object

⑤ Maintainability :- Objects can be maintained seperately Locating & fixing problems becomes easier

⑥ Reusability :- Objects can be reused in different Programs.

⑦ Design Benifits :-

OOP's make programmer to go through extensive planning, which makes better designs.

Once a program reaches certain size, OOP's are easy to program.

⇒ Disadvantages of OOP's :-

① Size :- Programs are larger than other Programs

② Effort :- Requires lot of work to create

③ Speed :- Slower than other program due to large size

④ Not all programs can be modelled by objects model

⇒ Java Buzzwords :-

① Simple :- Java is made easy for Programmer to understand and use effectively

② Secure :- Java is made secure by confining applet for execution into an environment and prevent its acess to other parts

③ Portable :- The same code must work on all computers. Being architecture neutral, Java is portable

④ Object - Oriented :- In Java everything is in object., Java can be extended as it is object oriented.

⑤ Multithreaded :- Java enables us to write programs that can do many tasks simultaneously.

⑥ Robust :- Java makes an effort to prevent error prone situation by compile time / runtime check.

⑦ Architecture neutral :-

Java has Write Once Run Anywhere approach so it has longevity and Portability

⑧ Interpreted :-

Java enables to write cross platform programs which are compiled into an intermediate representation called Java Byte. Any Machine with JVM (Java Virtual Machine) can run this program.

⑨ High Performance :-

Java Bytecode was carefully designed so that it would be easily translated to native language by Just-In-Time (JIT) Compiler for very high Performance.

⑩ Distributed :-

Java is built for distributed environment of Internet as it handles TCP/IP Protocols and it also supports Remote Method Invocation which enables us to invoke a method across network.

⑪ Dynamic :- Java is designed to adapt to runtime evolution. Java carries run-time information that can verify & resolve access to objects during runtime.

=> Java v/s C and C++ :-

→ C has header files, Java has Packages

→ C support Pointers but Java do not have pointers

→ C program is compileed into Machine Code, but Java program is first converted to Bytecode then into Machine Code

→ C++ support Operator overloading. Java does not

→ C++ has multiple inheritence but Java does not

→ In Java import statement are used for including Packages instead of #include

→ Java does not have Structure, Union @ enum datatypes

→ Java does not use Goto

→ Java does not have destructor like C++

→ Java does not support Storage classes like Auto, External etc...

⇒ Java Runtime Environment :-

(*) The smallest set of executables, and files that constitute standard Java Platform

(*) It provides libraries, JVM and other components to enable applets (or) applications written in Java.

(*) Java Plug-in :- Enables applets to run in Popular browsers

(*) Java Web Start :- Deploys standalone applications over a network.

⇒ Java Development Kit (JDK) :-

(*) It is a package that includes a large number of development tools, hundreds of classes and Java standard library API's and methods.

(*) It contains JRE and development tools.

(*) JDK provides tools to integrate and execute applications and applets

① javac :- The Java Compilers, it compiles java source code to Java Bytecode

javac filename.java

⑦ java :- The Java Interpreter, it executes Java class files created by Java Compiler

java Classfilename.

③ javap :- The Java class file dissasmbeler. Its output depends on options used. It disassembles class files

javap [options] Classfilename.

④ jdb :- The Java debugger, helps you find and fix bugs in Java language Program

jdb [options]

⑤ javah :- The C header and Stub file generator, produces C header files and source files from Java Class

javah [options] classfilename

⑥ javadoc :- The Java API Documentation generator. It generates HTML pages of API documentation from Java Source files.

javadoc [options] [package | source.java]*

⑦ appletviewer :- The appletviewer command allows you to run applets outside a web browser

appletviewer [options] filename.html [URL's]

→ Java Virtual Machine :-

It is the the virtual machine that runs that Java byte code. It is the entity that allows Java to be Portable

Source Code —Java Compiler→ Byte Code —Java Interpreter→ Machine Code

⇒ Structure of Java Program :-

Documentation

Package Statement

Import Statement

Interface Statement

Class Definitions

main method Class {

// definitions

}

⇒ Control Statements :-

① The if statements :-

```
if (condition) statement ;
```

Here, condition is a boolean expression. If condition is true statement is executed or else it is bypassed.

⇒ Example :-

```
Class IfExample {
    Public static void main (String args[]) {
        int x, y ;
        x = 10;
        y = 20;
        if (x < y) System.out.println ("x is less than Y");
        x = x * 2;
        if (x == y) System.out.println ("x is equal to Y");
        x = x * 2;
        if (x > y) System.out.println ("x is greater than Y");
        if (x == Y) System.out.println ("You won't see anything");
    }
}
```

② for loop :-

```
for (initilization ; condition ; iteration)
        statement ;
```

→ Example :-

```
Class ForTest {
    Public static void main (String args[]) {
        int x ;
        for (x = 0 ; x < 10 ; x++)
            System.out.println (" 5 * " + x + " = " + (x * 5));
        }
}
```

⇒ Blocks of Code :-

Java allows two (or) more statements to be grouped inside a block of code

Example :-

```
if (x < y) {  // Begin a block
    x = y ;
    y = 0 ;
} // End of block
```

=> Lexical Issues :-

① Whitespace :- blank, tab, space, newline

② Identifiers :- ⊗ Name given to variable, class, method

⊗ Can contain '_', '$', 0-9, alphabets

⊗ Should not begin with a number

Ex :-

temp123 , Int_18 , Age$ => Allowed

12ab → Not allowed => high-temp

③ Literals :-

A constant value is java is created by using a literal representation of it.

Ex :- 100, 98.6 , 'x'

④ Comments :-

(i) // This is Java Program

(ii) /* This is a comment
Example */

(iii) /** documentation */ (Doc comment)

javadoc tool uses doc comment when preparing automatic generated documentation

⑤ Seperators :-

(i) Paranthesses ( ) :- → Contain list of arguments
→ contain expression conditions

(ii) Braces { } :- Define code block, method, classes

(iii) Brackets [ ] :- declare array types

(iv) Semicolon ; :- Terminate Statements

(v) Comma , :- Seperate consecutive identifier and chain statements together inside for loop

(vi) Period . :- Seperate variable ~~name~~ ~~from~~ (or) method from reference variable

⑥ Keywords :-

* There are '50 Keywords' in Java
* These cannot be used as identifier
* True, False, null are also keywords.
* const, goto are reserved keywords but are not used

⇒ Datatypes in Java :-

① Primitive / Simple :-

(i) byte → 1 byte
(ii) short → 2 bytes
(iii) int → 4 bytes
(iv) long → 8 bytes

(i) to (iv) → Integers

(v) float → 4 bytes → Single Precision
(vi) double → 8 bytes → Double Precision

(v) & (vi) → Real / Decimal numbers

Integers + Real / Decimal numbers → Numbers

(vii) Char → 2 bytes → Characters

(viii) boolean → True (or) False

② Reference Data Types :-

- Arrays
- Strings
- Objects
- Interfaces etc...,

✱ Java does not support unsigned

✱ All math functions such as sin(), sqrt() return double values

⇒ Literals :-

① Integer Literals :-

* Integer literal can be assigned to byte, short, int & long variables

* To assign to a long variable (L or l at end)

```
long x = 9234578 1345 L
```

* To represent Octal value it must have leading zero

* To represent a Hexadecimal number it must be preceded with 'zero x' or '0x'

② Float literal :-

* Exponent is indicated by 'E' or 'e'

```
6.022E23
```

* In Java Floating point literal is default to double precision so it must be specified by F or f at end

```
float Pi = 3.14 F
```

* To specify Double there must be D or d at end

③ Character literals :-

It is represented inside a pair of single quates

'x' , 'A'

④ String literals :-

\'	→	Single quate
\"	→	Double quote
\ddd	→	Octal Character
\uxxxv	→	Hexadecimal Unicode Character
\\	→	Backslash
\r	→	Carriage return
\n	→	new line (Line feed)
\f	→	Form feed
\t	→	Tab
\b	→	Backspace

⇒ Variables :-

A basic unit of Storage

Syntax :-

type identifier [= value] [, identifier [=value] ...];

int a,b,c //Declare three variables of integer type

```
byte z = 22; // Initializes z
double pi = 3.14159; // Declares approximation of pi
```

⇒ Dynamic Initialization :-

Java allows variables to be initialized dynamically, using any expression valid at time variable is declared.

Ex :-

```
class DynInt {
    Public static void main (String args[]) {
        double a = 3.0 , b = 4.0;
        double c = Math.sqrt(a*a + b*b); // c initialized dyn
        System.out.println ("Hypotenuse = " + c);
    }
}
```

→ Scope of Variable :-

* A block defines a scope, every time block is started a new scope is created

* Two major scope in Java

  (i) Defined by Class

  (ii) Defined by Method

(x) The variables declared inside a scope are not visible to code outside the scope.

(x) When nested scopes are present i.e outer scope encloses inner scope, The elements (or) objects in outerscope are visible for inner scope but reverse is not true.

⇒ Type Conversion and Casting :-

① Widening Conversion / Automatic Conversion :-

→ It takes place only when

(i) Two types are compatible

(ii) The destination type is larger than Source type

→ Integer and Floating-point numbers are compatible with each other

→ Char, boolean and int are not compatible.

② Narrowing Conversion / Casting :-

To create a conversion between 2 incompatible types.

(target-type) value;

Example :-

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

* If integer value is greater than range of a byte, it will reduced modulo i.e the remainder of integer division by byte's range

* Truncation :- Type conversion that occurs when a floating point number value is assigned to integer type.

→ Automatic Type promotion :-

byte, short, char → int → long
↓
float
↓
~~Operand~~ Double

If any one operand is long, float or Double then whole expression is converted to long, floaat or Double respectively.

⇒ Operators :-

Four Groups

(i) Arithmetic

(ii) Bitwise

(iii) Relational

(iv) Logical

(1) Arithmetic Operators :- (Aritmetic + Assignment + Increment - Decrement)

+	Addition
-	Subtraction
*	Multiplication
%	Modulo division
/	Division
++	Increment
--	Decrement

→ Assignment operators :-

var = var op expression ;

+=, -=, /=, %=, *=

② Bitwise Operators :-

~	Bitwise Unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Bitwise Right Shift
>>>	Bitwise Right shift zero fill
<<	Bitwise Left Shift
&=	

③ Relational Operators :-

→ The outcome of these operator is boolean

→ Equality Test & Inequality Test :- Any type

→ Greater & less Than : Integer, floating-point, character

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than & equal to
<=	Less than & equal to

Example :-

```
int a=4, b=1;
boolean c = a<b;
```

(4) Logical Operators :-

&	Logical AND
|	Logical OR
^	Logical XOR
||	Short-Circuit OR
&&	Short-Circuit AND
!	Short-Circuit NOT

→ Boolean Logical operators :- Only operates on boolean operands

→ Short-Circuit Logic Operators : && and || secondary versions of Boolen AND and OR operators (& and |)

(5) The ? Operator :-

This operator can replace certain types of if-then-else

Syntax

expression 1 ? expression 2 : expression 3

Ex :-

```
(ratio = denom == 0) ? 0 : num / denom ;
```

⇒ Operator Precedence :-

Postfix	() [] .	Left to Right
Unary	++ , -- , ! , ~	~~Left to~~ Right to Left
Multiplicative	/ , * , % .	Left to Right
Additive	+ , -	
Shift	>> >>> <<	
Relational	> , >= , <= , <	
Equality	== , !=	
Bitwise AND	&&	
Bitwise XOR	^	
Bitwise OR	|	
Logical AND	&&	
Logical OR	||	
Conditional	?:	Right to Left
Assignment	= , += , -= , /= , *= , %= , &= ...	Right to Left
Comma	,	Left to Right

⇒ The Selection Statements :-

① if - else :-

```
if (condition) statement 1;
else statement 2;
```

if condition is true, statement 1 executes
if condition is false, statement 2 executes

② nested if :-

One if statement is in target of another if statement

```
if (condition 1) {
    if (condition 2) {
        if (condition 3) Statement 1;
            else statement 2;
    }
    else statement 3;
}
else statement 4;
```

③ The if-else-if ladder :-

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

④ Switch :-

```
Switch (expression) {
  Case value 1:
    // statement sequence
    break;
  Case value 2:
    // statement sequence
    break;
  .
  .
  .
  Case value N:
    // statement sequence
    break;
  default:
    // default statement sequence }
```

=> Iteration Statements :-

① while :-

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

② do-while :-

```
do {
  // body of loop
} while (condition);
```

③ for (initialization ; condition ; iteration) {

```
  // body of loop
}
```

Ex :- (i) for (int n = 10 ; n > 0 ; n--)

(ii) for (a=1, b=4 ; a < b ; a++, b--)

-> Nested loops :-

```
for (i=0 ; i<10 ; i++) {
    for (j=0 ; j< 10 ; j++)
        System.out.print("*");
    System.out.println();
}
```

⇒ Jump Statements :-

① break :-

(i) Terminates a sequence of statement in a switch statement

(ii) Used to exit a loop

(iii) Used as civilized form of goto

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

② Continue :-

It causes control to be transferred directly to the conditional expression that controls the loop.

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

③ Return :-

The return statement is used to explicitly return from a method. It causes program control to return back to caller of method

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

=> Arrays :-

It is a list of like-typed variables reffered by a Common name.

-> 1D Arrays :- List of like typed variable

-> Multi-D arrays :- Arrays of arrays

⇒ One - Dimensional Arrays :-

→ General form :-

```
type var_name [];
```

→ To allocate memory :-

```
array-var = new type [size];
```

(or)

```
type array-var[] = new type[size];
```

Eg :- `month_days = new int[12];`

→ Declaring more than one arrays :-

```
type [] arr1, arr2, arr3;
type arr1[], arr2[], arr3[];
```

⇒ Multi - dimensional arrays :-

```
type arrvar[][] = new [s1][s2]
```

s1, s2 → size

→ Unit - 2

Introducing Classes

* Class is a logical construct upon which entire Java Program is built.
* Class defines the shape and nature of an object.
* It defines a new Data Type
* Class is the template for an object

=> Declaring / Creating a Class :-

```
Class ClassName {
    type instance-variable1;
    type instance-variable2;
    //..
    type instance-variableN;

    type methodname1 (parameter-list) {
     //body of method
     }
    type methodname2 (parameter-list) {
     //Body of Method
    }
     .
     .
     type methodnameN (parameter-list) {
      //Body of Method.
    }
}
```

→ Instance Variables :- The variables defined within a class are called Instance Variables.

→ Methods & variables defined inside a class are collectively called members of a class

⇒ Access Specifiers :-

(i) Public :- Member can be accessed by any other code

(ii) Private :- Members can only be accessed by other members of its class

(iii) Protected :- Applies only when inheritence is used. Involved.

(iv) default :- When no specifier is used, then by default class is public within its own package.

⇒ Declaring Objects :-

→ Two Steps :-

(i) Define a variable of class type. This variable do not define an object

(ii) Acquire an actual, physical copy of object and assign it to the variable.

* The new operator dynamically allocates memory for an object and returns reference to it. This reference is the address in memory. This reference is then stored in variable.

→ General Syntax :-

```
ClassName Object_Name = new ClassName();
```

(or)

```
ClassName Object_Name;
Object_Name = new ClassName();
```

* An object reference is similar to memory pointer but object reference cannot be manipulated like pointers.

→ Assigning Object Reference Variables :-

```
Box b1 = new Box(); //An object created to class Box() and reference stored in b1.
Box b2 = b1 // Reference assigned to b2
```

// Here actual object is not copied just the reference is copied and assigned to b2.

Now, both b1 and b2 refer to the same object.

⇒ Declaring Methods :-

→ General form :-

```
type name(parameter-list) {
    // Body of Method
}
```

- ✳ type specifies the datatype of value returned
- ✳ name is the actual name of method
- ✳ Parameter-list is a sequence of type and identifier pairs seperated by commas.
- ✳ Parameters are essential variables that recieve the value of the arguments passed to method when it is called.
- ✳ Methods that have a return type other than void return a value to the calling routine using the following form of return statement:

```
return value.
```

⇒ Recursion:-

It is the process that allows a method to call itself.

```
// A Sample Example
Class Factorial {
    int fact(int n) {
        if (n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

Class Recursion {
    Public static void main(String args[]) {
        Factorial f = new Factorial;

    System.out.println("Factorial of 3 is" + f.fact(3));
    System.out.println("Factorial of 4 is" + f.fact(4));
    }
}
```

=> Overloading of Methods :-

In Java, it is possible to define two (or) more methods within a same class with the same name, as long as their Parameter declaration is different

With this feature Java support Polymorphism

The type and/or the name of arguments are used as guide to determine which version of overloaded method is used.

-> Method Overloading by changing no of arguments :-

```
Class Calculation {
    void sum (int a, int b) {
        System.out.println (a+b);
    }
    Void sum (int a, int b, int c) {
        System.out. println(a+b+c);
    }
    Public static void main (String args[]) {
    Calculation Ob = new Calculation();
    Ob. sum (10,10,10); ob. sum (10,20);
}
```

⇒ Method overloading by changing type of argument:-

```
class Calculation{
    void sum(int a, int b){
        System.out.println(a+b);
    }
    void sum(float a, float b){
        System.out.println(a+b);
    }

    public static void main(String args[]){
        Calculation ob = new Calculation();
        ob.sum(10.5, 10.5);
        ob.sum(8, 17);
    }
}
```

When an instance variable is accessed by code that is not the part of the class in which these variables are defined, it must be done through an object, by use of dot ~~product~~ operator.

⇒ Methods with Parameters :-

-> When a method does not have any Parameters, its use becomes very limited for example,

~~void Square (int i~~

```
int square () {
    return 10*10;
}
```

-> This square function returns the square of 10 only.

If we use a parameter, we can generalize this function for all integers

```
int square (int i) {
    return i*i;
}
```

-> Here, square returns the square of a number whose value is passed to it.

→ Parameter :- It is a variable defined by the method which takes values.

→ Arguments :- It is a value passed to a method when it is invoked.

⇒Constructors :-

- It can be tedious to intialize all the variables in a class each time an instance is created.
- A constructor intializes an object immediately upon creation. Once defined, a constructor is automatically called when object is created before the new operator completes.
- Constructors have same name that of the class and do not have a return type because the implicit return type of a class constructor is itself.
- When you do not explicitly define a Constructor, Java creates a default constructor for class and all non-initialized instance variables will have zero, null or false for number, reference and boolean.

→ Syntax :-

```
Class-var = new classname();

Box mybox1 = new Box();
```

```
class Box {
    double width;
    double length;
    double height;

    Box() {
        width = 10;
        length = 10;
        height = 10;
    }
    double volume() {
        return width * height * length;
    }
}

class BoxDemo {
    public static void main (String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol = mybox1.volume();
        System.out.println ("Box-1 Volume : " + vol);
        vol = mybox2.volume();
        System.out.println ("Box-2 Volume : " + vol);
    }
}
```

} Constructor for the class Box
It initiages the values of dimensions
when mybox1 and mybox2 are created.

-> Constructors can also be parameterized to make it more useful. In the previous page we observe that for both myBox1 and myBox2, dimensions intialized are the same i.e 10. Instead, we can add parameters to constructor and pass arguments to it while ~~call~~ invoking it.

=> Overloading Constructors :-

Constructor methods can also be overloaded with use of different number of parameters.

```
class Box{
    double width;
    double height;
    double length;

    Box(double w, double l, double h){
        weight = w;
        length = l;
        height = h;
    }
```

} Constructor with all dimensions.

```
Box () {
    width = -1;
    length = -1;
    height = -1;
}
```
} Constructor with no dimensions

```
Box (double len) {
    width = length = height = len;
}
```
} Constructor with one dimension for cube

```
double volume () {
    return width * length * height;
}
}

Class OverloadCons () {
    Public static void main (String args[]) {
        Box mybox1 = new Box (10, 15, 8);
        Box mybox_2 = new Box ();
        Box mybox_3 = new Box (8);

        double vol = mybox1.volume ();
        System.out.println ("Volume -1 : " + vol);
        vol = mybox2.volume ();
        System.out.println ("Volume -2 : " + vol);
```

```
    vol = mybox3.volume();
    System.out.println("Volume -3 :" + vol);
  }
}
```

→ Output :-

Volume -1 : 1200.0

Volume -2 : -1.0

Volume -3 : 512.0

⇒ Passing Objects as Parameters :-

In Java, just like the simple data types objects can also be used as Parameters to methods.

```
class Test {
    int a,b;
    Test (int i, int j){
        a=i;
        b=j;
    }
    boolean equalTo (Test ob) {
        if (ob.a == a && ob.b == b) return true;
        else return false;
    }
}
```

```
class PassObt
    public static void main (String args[]){
        Test ob1 = new Test(100,22);
        Test ob2 = new Test(100,22);
        Test ob3 = new Test(11,-8);

    System.out.println ("ob1=ob2 :" + ob1.equalTo(ob2));
    System.out.println ("ob2== ob3:" + ob2.equalTo(ob3));
    }
}
```

Output :

ob1 == ob2 : True

ob2 == ob3 : False

→ Argument Passing :-

→ Call by Value :- It is the type of invoking ~~objects~~ methods by passing the values. The methods makes a copy of the passed values. Any changes made to the variables in the Method does not effect outside method.

→ Call By Reference :- This is the type of invoking a Method in which the objects are passed as parameters into methods. Any changes made to the variables inside the method will also change outside the method.

→ Returning Objects :-

Methods can return any type of values even it can return an object.

```
Class Test {
    int a;

    Test (int i){
        a=i;
    }

    Test incrByTen (){
        Test temp = new Test (a+10);
        return temp;
    }
}

Class ReturnOb{
    public static void main (String args[]){
```

```
    Test ob1 = new Test(2);
    Test ob2;
     ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a:" + ob2.a);
  }
}
```

=> Output :-

ob1.a : 2

ob2.a : 22

→ The this keyword :-

- 'this' keyword can be used inside any method to refer to the current object.
- 'this' is always the reference to the object on which the method was invoked.

→ // A redundant use of this

```
Box(double w, double l, double h) {
    width = w;
    height = h
    length = l;
```

-> 'this' keyword can be used to resolve naming collision between instance variables and local variables

```
Box (double width, double length, double height) {
    this.width = width;
    this.length = length;
    this.height = height;
}
```

⇒ Static Variables and Methods :-

-> The static members declared in a class are just like global variables @ methods.

-> The static members are independent of any object. The static members are accessed without any reference to objects.

main() must be declared static because it must be called before defining any object.

-> Restrictions of Static Methods :-

(i) They can only directly access static method of their classes

(ii) They can only directly access static variables of their classes

(iii) They cannot reffer to this @ super

```
Class UseStatic {
    static int a, int b;
    a = 3;

    static void meth (int x) {
        System.out.println ("x = " + x);
        System.out.println ("a = " + a);
        System.out.println ("b = " + b);
    }

    static {
      b = a * 4;
    }

    Public static void main (String args[]) {
        meth (42);
    }
}
```

=> Varargs : Variable Length arguments :-

-> Beginning with JDK 5, Java included a feature that simplifies the creation of methods that need to take a variable number of arguments.

-> This method is called varargs short for variable length arguments.

-> A method that takes variable number of arguments is called a variable-arity method.

-> A variable-length arguments is specified by three periods (...)

-> A method can have normal parameters along with variable length parameters however variable length parameter must be declared last.

```
int doIt (int a, int b, double c, int... vals) {
```

-> The above example is completely acceptable

-> A method cannot have more than one variable length parameter.

```
int doIt (int a, int... vals, boolean... vals) => Invalid
```

=> Command line arguments :-

-> The Java Program can be run by passing the arguments needed for Program in the command line itself. These are called Command Line Arguments.

-> Command Line arguments are stored as string in the array args[] of the main method.

```
class Commandline {
    public static void main (String args[]) {
        for (int i = 0; i < args.length; i++)
            System.out.println ("args[" + i + "] : " + args[i]);
    }
}
```

→ Execution :-

java Commandline This is a test 100 -1

→ Output :-

args[0] : This

args[1] : is

args[2] : a

args[3] : test

args[4] : 100

args[5] : -1

→ The final Keyword :-

→ A field can be declared final. Doing so prevents it from being modified, making it essentially, a constant.

```
final int FILE_NEW = 1
final int FILE_OPEN = 2
final int FILE_SAVE = 3
final int FILE_SAVEAS = 4
final int FILE_QUIT = 5
```

- Common convention is to use uppercase letters for identifiers of final fields.
- The final keywords can also be applied to methods, method parameters and local variables.

→ Garbage Collection :-

→ In Java the objects that are dynamically allocated are destroyed and memory is reallocated after the execution automatically This process is called Garbage Collection.

→ When there is no reference to an object exist, that object is assumed to be no longer needed and the memory occupied by that object is reclaimed.

⇒ Nested and Inner Classes :-

- It is possible to define one class within another class, such classes are called as nested classes.
- The nested class has access to all the members of the enclosing class but The enclosing class does not have access to the nested class members.
- The nested class that is declaring directly within its enclosing class scope is a members of its enclosing class.

→ There are two types of nested classes :-

(i) Static Class :- It is the one that has static modifier applied, as it is static, it must access the non static members of enclosing class through objects only. It cannot refer to non-static members of enclosing class directly.

(ii) Non-static / Inner Class :- It has access to all methods, variables of the outer class and it may refer to them directly without declaring objects.

```
class Outer {
    int outer_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }
    class Inner {
        void display() {
            System.out.println("outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

=> Strings :-

- In Java, String is a sequence of characters. Java does not implement strings as array of characters instead it implements them as objects.
- In Java, a string once created cannot be changed, hence called as immutable strings.
- When a String is created then its contents cannot be changed, although everytime you want to make changes, you declare new String objects to store the changed string.
- For those cases where mutable strings are required, they are created either using StringBuffer or StringBuilder classes.
- The java.lang package contains all three String classes i.e String, StringBuffer and StringBuilder.

String Constructors :-

(i) String s = new String();

→ Creation of instance of a String without any characters

(ii) String (char chars[])

→ Creation of a String from an array of characters

(iii) String (char chars[], int startIdx, int numChars)

→ Creation of a String from subrange of a Character array.

(iv) String (String strObj)

→ Construction of a string from another String.

(v) String (byte chars[])

(vi) String (byte chars[], int startIdx, int charsNum)

→ Creation of string from an array of bytes with and without subrange.

The typical String class on internet uses ~~an~~ byte to initialize a character in a String. So, String class has another constructor for formation of string from a array of bytes.

→ For mutable Strings :-

String (StringBuffer strBufobj)

String (StringBuilder strBuildobj)

⇒ String length :-

The length of a String is the number of characters that it contains.

int length();

Ex :-

```
char chars[] = {'a', 'b', 'c'}
String s = new String (chars);
System.out.println (s.length());
```

→ Output : 3

⇒ String literals :-

→ Java provides an easy way to create strings

```
Char char[] = {'a', 'b', 'c'};
String s1 = new String (chars);
String s2 = "abc" // Use of Strong Literal
```

→ Everytime String literals are used, a String object is created.

→ So, methods can be called directly on Strings quated in double qoutes.

⇒ String Concatenation :-

Java does not allow operators to be applied on Strings but only exception is that '+' can be used to concatenate two Strings.

```
String age = "9";
String s = "He is" + age + "years old.";
System.out.println(s);
```

Output :- He is 9 years old.

→ String can also be concatenated with other data types.

```
int age = 9;
String s = "He is " + age + " years old."
System.out.println(s);
```

Output : He is 9 years old.

→ Here, the integer 9 is first converted into String, then concatenated as normal Strings.

-> String Conversion and toString() :-

-> When Java converts data into its String representation during concatenation, it does so by calling one of the overloaded versions of string conversion method valueOf() defined by String.

-> ValueOf() is overloaded for all simple types and for type object.

-> For ~~example~~ simple types, valueOf() returns a string that contain the human-readable equivalent of the value with which it is called. For objects, valueOf() calls the toString() method on the object.

-> Every class implements toString() because it is defined by Object.

=>Character Extraction :-

-> The String class provides a number of ways in which characters can be extracted from a String.

-> The characters in the String are indexed with integer numbers starting from zero.

(i) charAt() :-

Used To extract a single character from String

```
char charAt(int where);
```

'where' specifies the index at which character has to be extracted.

```
Char ch;
ch = "abc".charAt(1);
```

ch is assigned with b.

(ii) getChars() :-

Used to extract more than one character from a String and store it in a Character array.

```
Void getChars(int sourceStart, int SourceEnd, char target[], int targetStart);
```

sourceStart :- Index from where the Substring starts

sourceEnd :- Index where the Substring ends + 1

target :- The target array in which characters are stored

targetStart :- The location in target[] array from where Characters have to be copied.

(iii) getBytes()

It is an alternative for getChars() that stores characters into array of bytes.

```
byte [] getBytes();
```

It uses default character-to-byte conversions provided by Platform

(iv) toCharArray()

It converts all the characters in a String object into a character array.

```
char [] toCharArray()
```

⇒ String Comparison :-

The String class provides many methods for comparison ~~conversion~~ of two strings.

(i) Equals() and equalsIgnoreCase() :-

```
boolean equals(Object str);
```

The String str is compared with the invoking object String. If they are equal, True is returned else False. This method is case sensitive.

boolean equalsIgnoreCase (String str)

This also compares the String object str with the method invoking String object.

If both strings are equal, it returns True else returns false.

This considers A - Z to be equal to a - z.

(ii) regionMatches() :-

-> This compares one specific region inside a String with a specific region inside another string.

boolean regionMatches (int startIndex, String str2, int str2StartIndex, int numChars);

-> It also has a overloaded version which ignores case while comparison

boolean regionMatches (boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars);

(iii) StartsWith() and endsWith() :-

-> Two more specialized form of regionMatches() are present in String Class.

a) StartsWith() determines wether a given String begins with a specified string

b) endsWith() determines wether a given String ends with a specified String.

→ General Forms :-

boolean startsWith (String str)

boolean endsWith (String str)

→ Another form of StartsWith() :-

It allows to specify a starting point

boolen startsWith(String str, int startIndex)

(iv) equals() vis '==' :-

→ The equals() compares the characters inside the String object

→ The == operator just determines @ compares two object references to check wether they refer to same instance.

(v) compareTo() :-

→ This operator compares the strings and returns which String is greater, less than @ equal.

→ A String is greater than other if it comes after other in dictionary order.

→ A String is less than other if it comes before other in dictionary order.

→ int compareTo(String str)

- < 0 ⇒ The invoking String is less than str
- > 0 ⇒ The invoking String is greater than str
- = 0 ⇒ The invoking string is equal to str.

→ Another version of compareTo() that ignores case is also present.

int compareToIgnoreCase(String str)

⇒ Searching ~~Ind~~ Strings :-

(i) indexOf() :- Searches for the first occurrence of a character or a substring

(ii) lastIndexOf() :- Searches for the last occurrence of a character or a substring.

→ General form :-

```
int indexOf(char ch);
int lastIndexOf(char ch);
```

→ For substring :-

```
int indexOf(String str);
int lastIndexOf(String str);
```

-> To specify a starting point from where Search has to begin

int indexOf (char ch, int startIndex)

int lastIndexOf (char ch, int startIndex)

-> For Substring :-

int indexOf (String str, int startIndex)

int lastIndexOf (String Str, int StartIndex)

=> Modifying Strings :-

(i) substring() :-

-> It is used to extract the substring from a given String

String substring (String int startIndex)

-> It only takes the starting position from where substring has to be extracted

String substring (int startIndex, int EndIndex)

-> It takes both starting and ending position between which substring is extracted.

(ii) concat() :-

This method concatenates two strings and assigns the new concatenated string to a new String object.

```
String S1 = "Hello"
String S2 = S1.concat("Everyone");
```

(iii) replace() :-

It is a method used to replace all occurences of a character with another character.

String replace (char original, char replacement)

→ To replace a character sequence :-

String replace (CharSequence original, CharSequence replacement)

(iv) trim() and strip() :-

trim() method returns copy of the invoking string from which leading and trailing spaces have been removed.

String trim()

Ex:- String S = " Hello World ".trim();

This puts "Hello World" into S.

→ strip() method removes all whitespace characters from beginning and end of invoking String and returns the result.

```
String strip();
```

⇒ StringBuffer Class :-

→ The StringBuffer represents a growable and writable character sequence.

→ StringBuffer automatically makes room for any addition of new characters, to allow them to grow.

→ StringBuffer Constructors :-

(i) StringBuffer().

It reserves 16 characters room

(ii) StringBuffer(int size)

The number of characters are explicitly declared

(iii) StringBuffer (String str)

The String str sets the initial contents of StringBuffer and also room for 16 more characters are reserved.

(iv) StringBuffer (CharSequence chars)

It creates a StringBuffer from a character array.

(i) length() and capacity() :-

-> length() method determines the current length of StringBuffer

-> capacity() determines the total allocated capacity.

Ex:- StringBuffer sb = new StringBuffer("Hello");

length = 5

capacity = 21

(ii) charAt() and setCharAt() :-

-> charAt() gets the value of a single character from StringBuffer

-> setCharAt() is used to set the value of a character within a StringBuffer

char charAt(int where)

void setCharAt(int where, char ch)

(iii) getChars() :-

To copy a substring of a stringBuffer into an array, this method is used.

void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)

(iv) append() :-

This method concatenates the string representation of any other data type to the end of invoking StringBuffer

General forms :-

StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)

(v) insert() :-

This method inserts one String into another. Its overloaded forms accept int, char, Object, CharSequence etc..

→ General forms :-

StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object ob)

(vi) reverse() :-

This method returns the reversed characters of a Object StringBuffer on which it was called

StringBuffer reverse();

(vii) delete() and deleteCharAt() :-

-> The delete() method deletes a sequence of characters from the invoking object

-> The deleteCharAt() deletes only one character from the specified position.

StringBuffer delete(int startIndex, int EndIndex)

StringBuffer deleteCharAt(int Index)

(viii) replace :-

~~All occurrences~~ One set of characters can be replaced with another set inside a StringBuffer

StringBuffer replace(int startIndex, int endIndex, String str)

(ix) substring() :-

A portion of a StringBuffer can be obtained by calling this method

StringBuffer substring(int startIndex)

StringBuffer substring(int startIndex, int endIndex).

=> String Builder :-

-> StringBuider is just like StringBuffer but it is not synchronized, which means that it is not thread-safe.

-> The advantage of StringBuilder is faster performaces.

-> However, in cases in which a mutable string will be accessed by multiple threads and no external synchronization is emploged, it is best to use StringBuffer.

Unit-3 :- Inheritance

→ Inheritance is a special feature of Java that allows us for code reusability.

→ The properties of class can be inherited by another class and it may have its own property.

→ The class that extends its property to another class is called Superclass.

→ The class that inherits properties from other classes is called Subclass.

→ In Java, one Superclass can have many Subclasses called Heirarchial Inheritance but one subclass cannot have more than one Superclass.

→ No class can be superclass of itself

→ Although, a subclass of a Superclass can be a superclass for another Subclass.

→ Creation of Subclass :-

```
Class subclass-name extends superclass-name {
    //body of the class
}
```

Ex :-

```
Class A {
    int i;
}
Class B extends A {
    int j;
}
```

→ Now, class B contains two members i from superclass A and j of its own

→ Although a subclass includes all the members of its superclass it cannot access those members of the superclass that have been declared private.

→ A class member declared private remains private to its class. It is not accessible to any code outside its class including subclasses.

→ A major advantage of inheritence is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.

→ A reference variable assigned to a superclass can be assigned reference to any subclass derived from that Superclass.

→ When, a reference to a subclass object is assigned to a Superclass reference variable, you will have access only to the parts of objects defined by Superclass.

→ The Superclass has no knowledge of what subclass adds to it.

=> Using super :-

→ super is the keyword used in a subclass to refer to its immediate superclass.

→ super has two general forms :-

(i) → To call the superclass constructors

(ii) → The second is used to access a member of superclass hiddem by a member of subclass

→ A subclass can call a constructor defined by its superclass by the use of this general form.

```
super(arg-list);
```

→ Second use of super :-

```
super.member
```

The member can be a method (or) an instance variable.

→ Creating a Multi-level heirarchy :-

```
class Karnataka{
    int district;
}
class Mysuru extends Karnataka{
    int Taluk;
}
class Nanjangudu extends Mysuru{
    int Hobli;
}
```

⇒Method Overriding :-

→ In a class heirarchy, when a method in a subclass has same name and type signature as a method in superclass, then the method in subclass is said to override the method in the superclass.

→ When a overridden method is called from within its subclass, it always refers to version of method defined by the subclass. The version of method in Superclass is hidden.

→ Method overriding occurs only when the names and type signatures of the two methods are identical.

→ If they are not, then the method is simply overloaded.

⇒ Dynamic Dispatch of Method :-

→ Dynamic Method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

→ Dynamic Method dispatch is important in Java because it is how runtime Polymorphism is implemented.

→ When an overridden Method is called through a superclass reference, Java determines which version of that method to execute based upon the object being reffered to at the time call occurs.

→ It is type of object being reffered to that determines which version of an overridden method is executed.

→ Using Abstract Classes :-

→ Sometimes we have to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of methods that subclasses must implement.

→ You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.

→ These type of ~~abs~~ methods are sometimes called subclasser responsibility because they have no implementation specified in superclass.

→ General form to declare abstract Method:-

```
abstract type name (parameter-list);
```

→ Any class that contains one (or) more abstract method must also be declared abstract.

→ It is done as:-

```
abstract class class-name {
} //body of class
```

→ There can be no objects to abstract classes, because an abstract class is not fully defined.

→ Any subclass of an abstract class must either implement all of the abstract method in Superclass, (or) be declared abstract itself.

⇒ Using final with Inheritance:-

→ final can be used for 2 purpose:-

(i) Prevent Method Overriding

(ii) Prevent Inheritance

(i) Method overriding can be prevented with the use of final.

» A method declared final cannot be overridden

» The methods declared final sometimes provide performance enhancement : because the compiler knows that the method will not be overridden.

» The declaration of a method final causes early binding.

(ii) The inheritance of a class can be prevented by declaring it final.

» In A class declared final, all methods are also declared final.

» It is illegal to use both abstract and final declaration for same class.

=> The Object Class :-

» There is a special class in Java called Object class i.e superclass for all the classes.

» A reference variable of type object can refer to any other class.

⇒ Packages :- Packages are containers of classes

→ In Java, Package is both naming and visibility control Mechanism.

→ You can define classes inside a package that are exposed only to the members of same package.

→ To create a Java Package, simply include a 'package' command as the first statement in a Java Source File.

```
Package mypackage;
```

→ The 'package' statement defines a namespace in which packages are stored. If package statement is not present, then the classes are put into default package with no name.

→ Typically, Java uses file system directories to store packages, and ~~this~~ all .class files of the classes must be in the directy named mypackage for them to be part of mypackage.

→ More than one file can include the same package statement.

→ The 'package' statement simply specifies to which package the classes defined belong.

→ Heirarchy of Packages can also be created by simply seperating each package name from one above it by use of perid.

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

→ The package heirarchy must be reflected in file system of your JAVA development system.

→ For example, a package declared as

```
Package a.b.c;
```

needs to be stored in a\b\c in a Windows Environment

⇒ Finding Packages :-

(i) Java Run-Time system uses current directory as its starting point. Thus, if your package is a subdirectory of your current directory then it will be found.

(ii) You can specify a directory paths by setting CLASSPATH environment variable.

(iii) You can use -classpath option with javac and java to specify the path to your classes.

=> An Example :-

```
package mypack;

class Balance{
    String name;
    double bal;

    Balance (String n, double b){
        name = n;
        bal = b;
    }
    void show(){
        if (bal<0)
            System.out.print ("--->");
            System.out.println (" name + ": Rs" + bal);
    }
}
Class AccountBalance {
    public static void main (String args []) {
        Balance current[] = new Balance [3];
        current [0] = new Balance ("Sunil", 123.45);
        current [1] = new Balance ("Karthik", 157.02);
        current [2] = new Balance ("Rishab", -12.33);
        for (int i =0; i<3; i++)
            current[i].show();
    }
}
```

» Name the above file AccountBalance.java and put into a directory called mypack.

» Compile the file and make sure resulting .class file also in mypack directory.

» Execute AccountBalance class using the following command

java mypack.AccountBalance

⇒ Packages and Member Access :-

Packages add another dimension for access control. Packages acts as containers of Classes and Subpackages Classes acts as containers of Data and Code. The class is Java's smallest unit of Abstraction.

	Private	Default	Protected	Public
Same Class	Yes ~~Private~~	Yes	Yes	Yes
Same Package Subclass	No	Yes	Yes	Yes
Same Package Non SubClass	No	Yes	Yes	Yes
Different Package SubClass	No	No	Yes	Yes
Different Package Non SubClass	No	No	No	Yes

→ Anything declared public can be accessed from different classes and different Packages.

→ Anything declared private cannot be seen outside its class.

→ When a member does not have any access specifier i.e it has default then it is visible to subclasses as well as otherclasses in same package.

→ The members declared protected ~~can~~ is not visible to any other classes of different package. It is only visible to all classes inside the same package and only subclass in different Package.

→ There must be only one class specified as Public in a java file and file name should be same as that class name.

A Java Package created can be put into a destination by using -d

javac -d . Simple.java

This stores the Simple.class into current directory

⇒ Importing Packages :-

- To import a package in Java, it includes the import statement to bring certain classes (or) entire packages to visibility.
- Once imported a class can be directly called using its Name.

→ General Form :-

```
import pkg1 [.pkg2].(classname | *);
```

→ The above statement specifies hierarchy of packages. The * used in the end is to tell the compiler to import the whole package.

→ Ex :- `import java.util.*;`

→ The package java.lang is implicitly imported by the compiler in every program

→ When two classes (from 2 different packages) have same name, there will be a compile-time error. So you have to explicitly name the class specifying its package.

=> Inbuilt Java Packages and its classes :-

① java.lang :-

- String
- System
- StringBuffer
- Math

② java.util :-

- ArrayList
- LinkedList
- Date
- Calendar

③ java.io :-

- BufferReader
- File
- Inputstream
- OutputStream

④ java.awt :-

- Button
- Color
- Event
- Font

⑤ java.net :-

- Socket
- URL
- URLEncoder
- Authentecator

⑥ java.sql :-

- Connection
- DriveManager
- Resultset

⇒ Interfaces :-

- Using key word interface you can fully abstract a class from its implementation.
- Interfaces are just like classes but they lack the instance variables.
- Once an interface is defined, any number of classes can implement it.
- Also, one class can implement ~~and~~ more than one interface.
- Using Interfaces, we can implement "One interface, multiple Method" feature of Polymorphism.
- Interfaces are designed to support Dynamic method resolution at runtime

⇒ Why use Java Interfaces?

→ It is used to achieve fully abstraction

→ By Interface, we can support the functionality of multiple inheritance

→ It can be used to achieve loose coupling.

=> Defining an Interface :-

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

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

-> Interfaces are just like abstract classes but the Major difference is that :-

- Interfaces contain only Abstract methods and constants
- Abstract classes may contain instance variables and non-abstract methods.

- In the above declaration of an interface, the methods have no bodies.
- The classes implementing the interface must define all the methods declared in interface.

⇒ Implementing Interfaces :-

→ Once an interface has been defined, one @ more classes can implement that interface.

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

→ If a class implements more than one interface, the interfaces are seperated with comma.

→ The Methods that implement interface must be declared public. The type signature of the implementing method must exactly match the interface Method.

⇒ Nested Interface :-

An interface can be declared as a member of a class @ another interface. Such an interface is called a Member Interface @ nested interface.

A nested interface can be declared as public, private @ protected.

Example :-

```
class A {
    public interface NestedIF {
        boolean isNotNegative(int x);
    }
}

class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x<0? false : true;
    }
}

class NestedIFDemo {
    public static void main (String args[]) {
        A.NestedIF nif = new B();
        if (nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if (nif.isNotNegative(-12))
            System.out.println("This won't be displayed");
    }
}
```

⇒ Variables in Interface :-

→ Interfaces can be used to import shared constants into multiple classes by simply declaring an interface that contains variables that are initiazed to desired values.

→ If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything.

⇒ Default Interface Methods :-

→ A default method lets you define a default implementation for an interface method.

→ By use of a default method, it is possible for an interface method to provide a body, rather than being abstract.

→ The default Method is also reffered as an Extension Method.

⇒ Use Static Methods in an Interface :-

→ Like static methods in a class, a static method defined by an interface can be called independently of any object.

→ Thus, no implementation of interface is necessary and no instance of ~~instance~~ interface. is required

→ A Static method in a Interface is called as

InterfaceName.staticMethodName

Exception Handling :-

→ An exception is an abnormal condition that arises in a code sequence during runtime. In other words, exception is a run-time error.

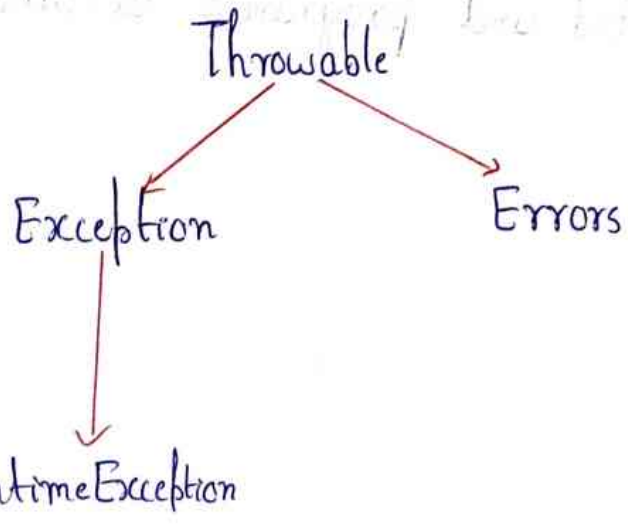
→ A Java exception is an object that describes an exceptional condition that has occured in a piece of code.

→ Types of Errors :-

(i) Syntax Error
(ii) Runtime Error
(iii) Logic Error

→ The exception can be generated by the Java Run-Time System (or) they can be manually generated by our code.

Exception Types :-



① → Checked Exceptions :-

→ All Exception types are subclasses of the built-in class Throwable.

→ Thus Throwable is at the top of the exception class hierarchy.

→ Throwable contains two subclasses 'Exception' and 'Errors'

→ The Exception class is used for exceptional conditions that the user programs should catch and also to create our own exception Types.

→ 'RuntimeException' is a subclass that has unchecked Exceptions

→ The Checked exceptions are checked by Java Compiler

→ If a checked exception is caught then exception handling code will be executed and program's execution continues.

② Unchecked Exceptions :-

→ All Exceptions that extend the RuntimeException or any one of its subclass are unchecked Exceptions.

→ They are unchecked by Compiler

→ To handle Unchecked Exception java interpreter will provide default handler. But in this case execution of program will be stopped.

→ Example :-

(i) Arithmetic Exception (Divide by 0)
(ii) Array Index Out of Bounds Exceptions
(iii) FileNotFound Exception
(iv) NullPointer Exception
(v) IllegalArgumentsException

⇒ try and catch :-

→ In Java Exceptions can be handeled by the programer himself

→ Two advantages of handling exception ourselves is

(i) Allows you to fix the error

(ii) Prevents program from automatically terminating

⇒ General Form :-

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

→ Once an exception is thrown program control transfers out of try block to catch clause. i.e catch is not called so control never returns back to try block from a catch.

→ The program execution continues to the next line after catch block.

→ A scope of catch clause is restricted to those statements specified by immediately preceeding try statement.

⇒ Example :-

```
Class Exc {
    public static void main (String args[]) {
        int d, a;
    } try {
        d = 0;
        a = 42 / d;
        System.out.println ("This will not be printed\n");
    } catch (ArithmeticException e) {
        System.out.println ("Division by Zero");
    }
    System.out.println ("After catch Statement");
}
}
```

⇒ Multiple Catch Clause :-

→ In some cases, more than one exception could be raised by a single piece of code.

→ To handle these, you can specify two or more catch clause, each catching a different type of execution.

→ When an exception is thrown each catch block is checked in order and first one whose type matches is executed.

→ After one catch statement executes, the others are bypassed and execution continues after try/catch block.

⇒ throw :-

→ In Java, it is possible to throw an exception explicitly using throw statement.

General form: throw ThrowableInstance;

→ Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

→ Two ways of obtaining a Throwable object

(i) Using a parameter in a catch clause

(ii) Creating one with the new operator.

⇒ Example :-

```
class ThrowDemo{
    static void demoproc(){
        try{
            throw new NullPointerException("Demo");
        }catch(NullPointerException e){
            System.out.println("Caught inside demoproc");
            throw e;
        }
    }
    public static void main(String args[]){
        try{
            demoproc();
        }catch(NullPointerException e){
            System.out.println("Recaught: "+ e);
        }
    }
}
```

⇒ throws :-

→ If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of this method can gaurd themselves with this exception.

→ A throws clause lists all the type of exceptions that a method might throw.

→ All methods that can throw exceptions except Runtime exceptions & errors are required to be declared in throws Clause. If not, a compile-time error will result.

⇒ General form:-

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

→ Here, exception-list is a comma seperated list of exceptions that the method can throw.

⇒ finally :-

→ finally creates a block of code that will be executed after a try/catch block has completed.

→ The finally block will execute wether or not an exception is thrown.

→ If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

→ The finally block is also executed just before the method returns.

→ The finally clause is optional. However each try statement must have at least one catch or a finally clause.

⇒ Java's Built In Exceptions. (java.lang) :-

→ Checked Exceptions

① ClassNotFoundException	Class not found
② IllegalAccessException	Access to a class is denied
③ InterruptedException	One thread has been interrupted by another
④ NoSuchMethodException	A requested method does not exist.

⇒ Unchecked Exception :-

(1) ArithmeticException → Division by zero

(2) ArrayIndexOutOfBoundsException → Array index is out of Bounds

(3) ClassCastException → Invalid Cast

(4) NullPointerException → Invalid use of null reference

(5) NegativeArraySizeException → Array created with negative size.

⇒ Creating Own Exception Subclasses :-

→ Own Exceptions can be created just by defining a subclass of Exception (which is a subclass of Throwable)

→ Exception does not have any methods but Throwable contains some defined methods.

→ Constructors of Exception :-

* Exception() ⇒ No description
* Exception(String msg) ⇒ Specify Description of Exception

⇒ Multithreaded Programming

→ A Multithreaded Program contains two (or) more parts that can run concurrently. These each part are called Threads.

→ Each Thread defines a seperate path for execution.

(i) → Process-Based Multi-Tasking :-

Two (or) programs are run simultaneously in a computer

(ii) Thread-Based Multi-Tasking :-

Two (or) more threads in a same program running simultaneously

⇒ Advantages of Multi Threaded Programming :-

(i) Multithreaded Multitasking control is under Java

(ii) Threads occupy the same address space

(iii) Inter thread communication is inexpensive

(iv) Multithreading enables us to write efficient programs to make maximum use of processing power available in system

(v) Multithreading helps you reduce the idle time because another thread can run while one is waiting.

→ It does not block the user because threads are independent and you can perform multiple operations at same time.

→ You can perform many operations together so it save time

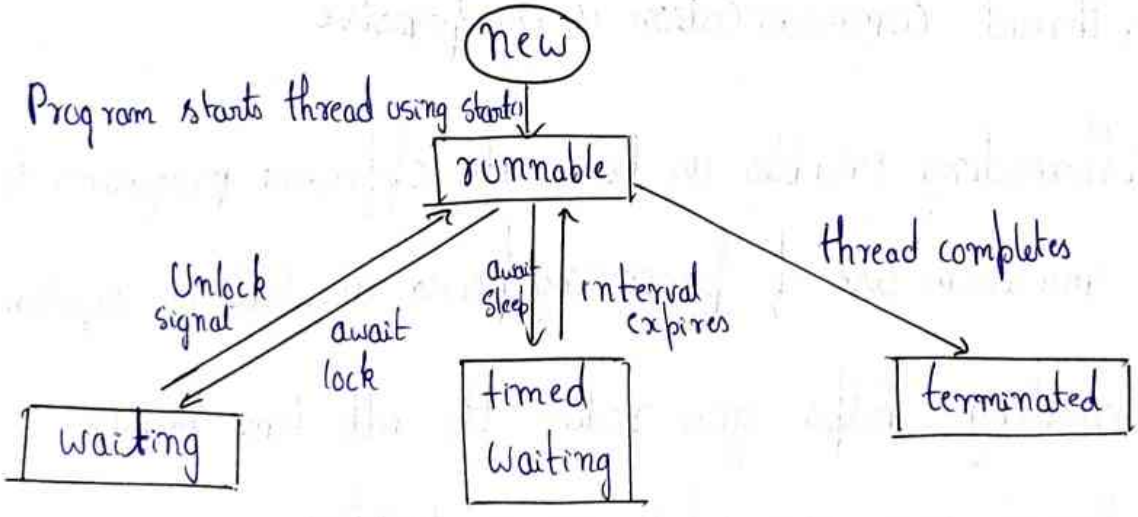
→ Threads are independent so it doesn't affect other threads if exception occurs in single thread.

⇒ The Java Thread Model :-

→ The java runtime depends on threads for many things and all class libraries are designed with multi threading in mind.

→ Java uses threads to enable the entire environment to be asynchronous which helps to reduce ineffeciency by preventing the waste of CPU cycles.

⇒ Life Cycle of a Thread :-



⇒ Thread exist in several states :-

* A thread can be 'running'
* It can be 'ready to run' as soon as it gets CPU time
* A running thread can be 'suspended', which temporarily suspends its activities
* A suspended thread can be 'resumed,' allowing it to pickup where it left off.
* A thread can be 'blocked' when waiting for a resource
* At any time, a thread can be 'terminated', which halts its execution immediately
* Once Terminated, a Thread cannot be resumed.

⇒ Thread Priorities :-

→ Java assigns to each thread a priority that determines how that thread should be treated with respect to others.

→ Thread priorities are integers that specify the relative priority of one thread to another.

→ An higher priority does not mean that the threads run faster instead it used to determine when to switch from one running thread to next. This is called a 'Context Switching'.

⇒ Rules of Context Switching:-

(i) A thread can voluntarily relinquish control:-

This occurs when explicitly yielding, sleeping or when blocked. In this case, all threads are examined and an highest-priority thread that is ready to run is given to CPU.

(ii) A Thread can be preempted by a higher-priority thread:-

A lower-priority thread ~~do~~ that does not yield the processor is simply prempted - no matter what it is doing - by a higher-priority Thread.

→ In cases where two threads have same priority are competing for CPU cycles, In some OS, they are time-sliced automatically in round-robin fashion.

→ For some other OS, threads of equal priority must voluntarily yield control to their peers.

⇒ The Thread Class and Runnable Interface :-

→ Java's Multithreading is based on Thread Class, its methods and its companion interface Runnable.

→ Thread encapsulates a thread of execution.

⇒ Thread Class Methods

- getName → Obtain a Thread's name
- getPriority → Obtain a Thread's Priority
- isAlive → Determine wether a thread is still running
- join → Wait for a Thread to Terminate
- run → Entry point for a Thread
- sleep → Suspend a thread for a period of Time
- start → Start a thread by calling its run method.

⇒ The Main Thread :-

→ When a Java Program starts up, one thread begins running immediately. This is usually called the main thread of your program because it is the one that is executed ~~from~~ when program begins.

→ The main Thread is important for 2 reasons :-

(i) It is the thread from which other "child" threads will be spawned

(ii) often, it must be the last thread to finish execution before it performs various shutdown actions.

→ Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.

→ To do so, a Thread object must be obtained calling the method currentThread()

```
Static Thread currentThread();
```

⇒ Creating a Thread :-

→ Two ways to ~~implem~~ create a Thread object

(i) Implementing the Runnable Interface

(ii) Extending the Thread class, itself.

① Implementing Runnable :-

→ Runnable abstracts a unit of executable code.

→ To implement Runnable, a class need to implement only a single method called run(), declared as

```
public void run();
```

→ Constructor of Thread class to created Thread Object

```
Thread (Runnable threadOb, String threadName)
```

```
Class New_Thread implements Runnable {
    Thread t;
  NewThread() {
      t = new Thread (this, "Demo Thread");
      System.out.println("Child Thread : " + t);
  }
  Public void run() {
      try {
          for(int i=5; i>0; i--) {
              System.out.println ("Child Thread : " + i);
              Thread.sleep(500);
          } catch (InterruptedException e) {
              System.out.println ("Child Interrupted");}
```

```
        System.out.println("Exiting Child Thread");
    }
}
class ThreadDemo1
    public static void main(String args[]){
        New_Thread nt = new New_Thread();
        nt.t.start();
        try{
            for(int i=5; i>0; i--){
                System.out.println("Main Thread : " + i);
                Thread.sleep(1000);
            } catch (InterruptedException e){
                System.out.println("Main Thread Interrupted");
            }
        System.out.println("Exiting Main Thread");
    }
}
```

② => Extending Thread :-

- Threads can also be created by extending the Thread class.
- The extending class must override run method, which is entry point for a new thread.

→ Constructor to create new thread :-

```
Public Thread (String threadName)

Class NewThread extends Thread {
  NewThread() {
    super ("Demo Thread");
    System.out.println ("Child Thread : " + this);
  }

  public void run() {
    try {
      for (int i=5; i>0; i++) {
        System.out.println ("Child Thread : " + i);
        Thread.sleep(500);
      } catch (InterruptedException e) {
        System.out.println ("Child Interrupted");
      }
```

```
        System.out.println("Exiting Child Thread.");
    }
}
class ExtendThread{
    public static void main(String args[]){
        New Thread
        NewThread nt = new NewThread();
        nt.start();
        try{
        for(int i=5; i>0; i--){
            System.out.println("Main Thread :"+ i);
            Thread.sleep(1000);
        } catch(InterruptedException e){
            System.out.println("Main Thread Interrupted");
        }
        System.out.println("Main Thread Exiting");
    }
}
```

Applet Programming

-> Applet is a small Java Program that is loaded into a Web Browser i.e it is embedded into a Webpage. It runs inside the Web Browser and works at Client side.

-> All applets are subclasses of java.Applet class

-> Applets do not have any main method

-> Applets are not standalone programs. They run either on web browser (or) applet viewer provided by JDK

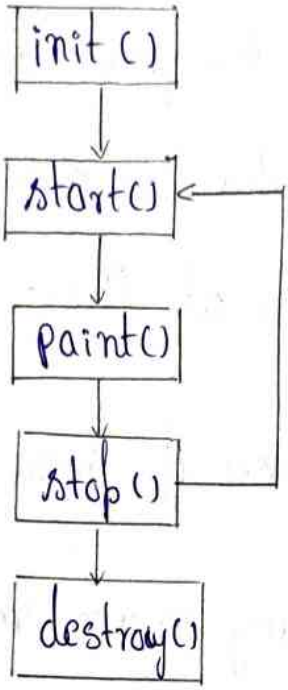
=> Advantages of Applets :-

-> It runs inside browser and works on client side so less response time

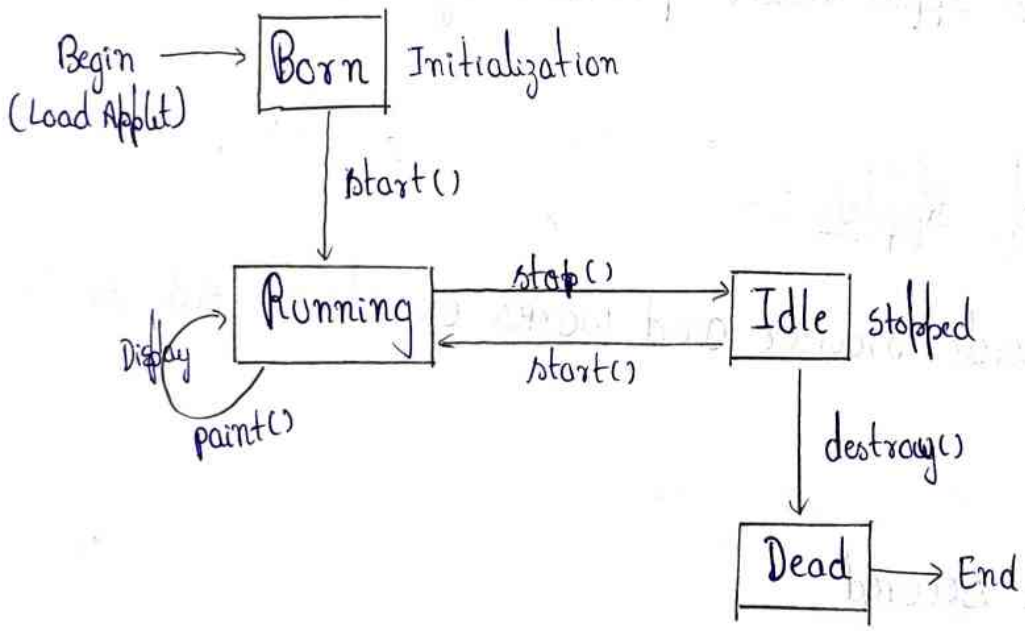
-> It is more secured

-> It can be executed by Multi-platforms with any browser i.e Windows, Linux, Mac Os etc...

⇒ Life Cycle of an applet :-



Order of methods of Applet class called



An Applet's State Transition Diagram

- init() :- (Born State)
  - This is the first method to be called
  - This is where all the variable should be initialized
  - This method is called only once during runtime of your applet

- start() :- (Running State)
  - This is the method called after init()
  - It is also used to start an applet that has been stopped @ to start a newly initialized applet.
  - Unlike init(), this method is called everytime an applet is resumed i.e when a user leaves a webpage and returns.

- paint() :- (Running State)
  - The paint() method is called each time an applet's output must be redrawn.
  - This method works when the applet is in Running state.

- stop() :- (Idle State)

When this method is called, the applet stops execution and goes into Idle state from where it can be resumed @ destroyed.

⇒ destroy() :- (Dead State)

* An applet from Idle state can be destroyed to end the execution of the applet using this method.

* The applet cannot be resumed from Dead State

⇒ Applets v/s Applications :-

(i) Applets do not use main method for initiating the execution of the code. Applets when loaded call certain methods of Applet class and begin execution.

(ii) Unlike standalone applications applets cannot run independent. They run inside a webpage with help of HTML tag.

(iii) Applets cannot read from @ write into files in local Computers.

(iv) Applets cannot communicate with other servers in network.

(v) Applets cannot run any program from local computer

(vi) Applets are restricted from using libraries from other languages such as C and C++. (But, Java language supports this feature through native methods).