Table of Contents:

---

10. Method Overriding · this Keyword:

- this calling statement:
- Differences:Static Members & Non-Static Members
- Differences Method Overloading & Method Overriding

11. Abstract class

- Concrete method:
- Concrete class:
- Abstract method:
- Abstract class:
- What is Abstract class?

12.4 java types(class,interface,annotation,enumeration)

- Annotation
- Enumeration

13. Interfaces

- Explain how multiple inheritance can be achieved through interface in java?
- What is Marker interface?
- Difference: Abstract class & Interface

14. Call by Value and Call by Reference

15. Casting(Type & Object)

- Homogeneous Statements:
- Heterogeneous Statements:
- Primitive Casting Widening Operation:
  Narrowing Operation:
- Object Casting
  Up-Casting
  Down-Casting

16. Polymorphism · Compile-Time Polymorphism. · Run-Time Polymorphism.

17. Abstraction

18. Encapsulation

19. Package, Access Levels and Imports

20. Design Patterns · Singleton class Design Pattern.

- Doubleton Class Design Pattern.
- Immutable Class Design Pattern

21. IIB

22. SIB

23. Object Class - (non-static method)

- toString()
- equals()
- hashcode()

24. Assert

25. String Class

- 2 types ways of creating String class Objects
- String Pool

26. Arrays

- Types of Arrays:
  Primitive Array: Derived
  Array:

27. Collection API

- List
- Queue
- Set
- Map

28. Wrapper Class - Boxing and UnBoxing

29. Exception Handling - 3 categories of exceptions:

- Java Unchecked Run-time Exception.
- Java Checked Exceptions: - Exceptions Methods: - 2 types of Exceptions

30. File Handling

31. Java Memory Management - Garbage Collection

32. Final, Finally, Finalize

33. Multi Threading

- Thread
- Thread will have 3 properties
- Thread Synchronization

34. Generics

35. Inner class

- Local Inner Class.
- Static Inner Class.
- Anonymous Inner class.
- Differences: Local Inner Class & Static Inner Class

36. Interview Example(Common)

37. Applet class

38. Event Handling

1. INTRODUCTION:

Installing JDK

There are 3 stages of Java Application development,
1) Coding/Composition
2) Compilation 3) Execution

1) Coding: Developing a program according to java syntax(rules) is called as Coding. Note:All the java programs should be saved with file-name.java extension.

2) Compilation:Converting the java program written in high level language into byte code by using java compiler is called as Compilation.
Note: All the byte code will be stored as .class Extension

3) Execution: Converting the byte code into machine level language or processor understandable instructions using JVM is called as Execution(JVM is an Interpreter).

Steps to set the java path in System Environment variable path:

1) Click on start, My Computer, Go to : C:¥Program Files¥Java¥jdk1.8.0_102¥bin and copy the path.
2) Click on start, right click on My Computer, click on properties.
3) Click on Advanced System variables settings, Environment variables.
4) Select the variables path under System variables section and click on edit button.
5) Under Variables value field, Go to end of the line and put a semi colon(;).
6) Paste the complete java path, click on OK.

Note: Java is case sensitive programming language.

Tokens

Tokens are smallest unit of a program. There are 3 important tokens, 1) Keywords.
2) Literals.
3) Identifiers.

1) Keywords: This are pre-defined words in the programming language which will have some pre-defined meaning.

Example: class, public, static, void, etc., There are approximately 50 keywords.
Rules to Write keywords: All keywords must be written in lower case.

2) Identifiers: Programmer defined words are called as Identifier.
Example: Name of variable, Name of a class, Name of a package, Name of an interface.

Rules to write Identifier:

A) Identifier can be alphanumeric, no special characters are allowed except $, &, -.
B) Identifier cannot start with a number but can start with a alphabet.
C) Space is not allowed.
D) Keywords cannot be Identifier.

3) Literals: Literals are values used in the programs. There are 4 literals, A)
Numeric Literals:

Any number represented in java program will fall under numeric literals category. B)
Character Literals:

Any character of the keyboards enclosed within single quote and 1 character within single quote is called as character literals. Example: 'A', 'I','9'.

C) String Literals:

Anything which is enclosed within double quotes are considered as String literals.
Example: "hello", "hello123". D)
Boolean Literals:

There are only 2 Boolean literals True and False. To take decision we will use Boolean literals.

Note:

1)True and False are also called as Reserved words.
2)0 and 1 are not Boolean literals.
3)Using System.out.println();, we can print Literals

Rule:

1) While writing java program, programmer should follow the condition of java programming language if the programmer is not following rules and conditions of java programs then java compiler will deploy Compile Time Error(CTE).
2) If the program results in compile time error, .class file won't be generated.

Explain how java is platform independent?

Java is platform independent because java programs are compiled in one java can be executed directly in other O.S without recompiling it, by just installing JVM which is part of JRE package.
JRE(Java Run-Time Environment) is a package containing,

1) JVM
2) Platform supporting files.
3) Standard Libraries of Java

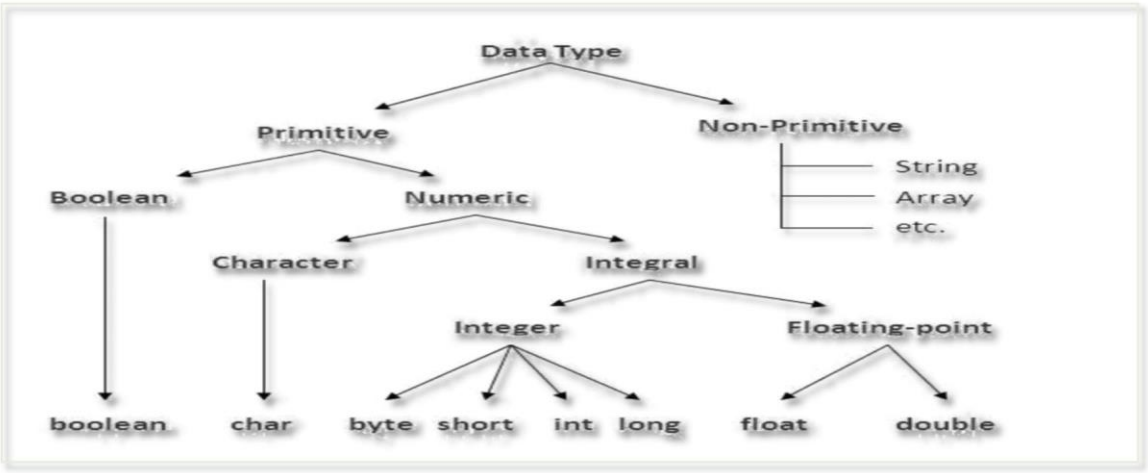
Whenever we compile a Java Program using java compile we will get the byte code which is also called as Intermediate code which can be understood by JVM, i.e., byte code is not with respect to and Platform it is with respect to JVM.
JVM will convert the byte code into underlying platform standards on which it is installed.

2. DATA TYPES & VARIABLES

Data Types

Java is a strongly typed language. This means that every variable must have a declared type. There are eight primitive types in Java.

1) Four of them are integer types.
2) Two are floating-point number types.
3) One is the character type char. 4) Boolean type for truth values.



1) Integer Types: The integer types are for numbers without fractional parts. Negative values are allowed.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type.

Name	Width	Range
Long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Int	32	–2,147,483,648 to 2,147,483,647
Short	16	–32,768 to 32,767
Byte	8	–128 to 127

1) byte:

1) The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127.
2) Variables of type byte are especially useful when you're working with a stream of data from a network or file.
3) They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
4) byte variables are declared by use of the byte keyword. **Example:** byte b, c;

2) short:

1) short is a signed 16-bit type. It has a range from –32,768 to 32,767.
2) It is probably the least-used Java type. **Example:** short s; short t;
3) **int:**

1) The commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
2) Variables of type int are commonly employed to control loops and to index arrays.

When byte and short values are used in an expression they are promoted to int when the expression is evaluated.

4) long:

1) long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.

2) The range of a long is quite large. This makes it useful when big, whole numbers are needed. **Example:** class Light

```
{
    public static void main(String args[])
    {
        int lightspeed; long days; long seconds; long distance;
        lightspeed = 186000; days = 1000; // specify number
        of days here seconds = days * 24 * 60 * 60; // convert
        to seconds distance = lightspeed * seconds; // compute
        distance
       System.out.print("In " + days+ " days light will travel about " +distance + " miles.");
    }
}
```

Output: In 1000 days light will travel about 16070400000000 miles.

---

2) Floating-Point Types: The floating-point(also known as real numbers,) types denote numbers with fractional parts. They are used when evaluating expressions that require fractional precision. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively.

5) float:

1) The type float specifies a single-precision value that uses 32 bits of storage.
2) Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. 3) Variables of type float are useful when you need a fractional component. **Example:** float hightemp, lowtemp;

6) double:

1)Double precision, as denoted by the double keyword, uses 64 bits to store a value.
2)Double precision is actually faster than single precision.
3)All transcendental math functions, such as sin( ), cos( ), and sqrt( ) return double values.
Example: program that uses double variables to compute the area of a circle:

```
class Area { public static void main(String
    args[]{ double pi, r, a; r = 10.8; // radius
    of circle pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

---

3) Character-Types:

1) The data type used to store characters is char.char is a 16-bit type.
2) The range of a char is 0 to 65,536. There are no negative chars.

3) The standard set of characters known as ASCII still ranges from 0 to 127 and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

Example:

```
class CharDemo
{
    public static void main(String args[])
    {
        char ch1, ch2; ch1 = 88; //
        code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Output:

```
ch1 and ch2: X Y
```

---

4) Boolean:

1) Java has a primitive type, called Boolean, for logical values.
2) It can have only one of two possible values, true or false. T
3) his is the type returned by all relational operators, as in the case of a < b.
4) Boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

Example:

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

Output:

```
b is false b is
true This is executed.
10 > 9 is true
```

----------Variables

The variable is the basic unit of storage in a Java program.
A variable is defined by the combination of a type, an identifier and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

There are 2 Variables data-types,

1) Reference Variables/Non-Primitive Variables data-type:
   This are used to store address.
   They are any instantiable class as well as arrays Ex:
   String, Scanner, Random, Die, int[], String[], etc.

2) Primitive Variables data-type:
   This are used to store primitive data.
   Ex: byte, short, int, long, float, double, boolean, char

There are 3 stages in variables usage:

1) Declaration.
2) Initialization. 3) Utilization.

1) Declaring a Variable:

a) In Java, all variables must be declared before they can be used.
b) Declaration means Creation.
c) Syntax:

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

type: is one of Java's atomic types, or the name of a class or interface.
Identifier: is the name of the variable.
Note: To declare more than one variable of the specified type, use a comma separated list.

2) Initialization:

a) It means storing a value inside the varibales.
b) Syntax:

```
Variables name=value;
```

Example: A=90;

3) Utilization:

a) Using the value stored in the variable for some purpose is called as Utilization.
b) Using System.out.println(); we can print the value stored in the variables.

Example:

```
public class Simple1
{
    public static void main(String[] args)
    {
        int stdId;        //Declration double
        stdPercent;
```

```
        boolean passed; stdId=10;
                //Initialization
        stdPercent=99.9; passed=true;
        System.out.println(stdId); ////Utilization
        System.out.println(stdPercent);
        System.out.println(passed);
    }
}
```

Output:

```
10 99.9 true
```

Example: class

```
Simple2
{
    public static void main(String[] args)
    {
        int empId; double
        empSal; char
        empGrade;

        empId=7; empSal=900000;
        empGrade='A';

        System.out.println("Employee Id:" +empId);
        System.out.println("Employee Salary:" +empSal);
        System.out.println("Employee Grade:" +empGrade);
    }
}
```

Output:

```
Employee Id:7
Employee Salary:900000.0 Employee
Grade:A
```

Example: class

```
Simple3
{
    public static void main(String[] args)
    {
        int a;//Declaration a=90;//Initialization
        System.out.println(a);//Utilization
        System.out.println(a);//Re-itilization
    }
}
```

Output:

```
90
190
```

Example: class
Simple7
{
    public static void main(String[] args)
    {
        int a=90; int
        a=190;
        System.out.println(a);
    }
}

Output:

Uncompilable source code - variable a is already defined in method main(java.lang.String[]) **Note:**

1) Primitive variables: The variables which are declared using any one of the 8 primitive data type are called as Primitive variables.
2) Strng is not a primitive data type, it is a Bult-in class.

Example: int a; double d;

3) We cannot create 2 variable with the same name, inside a same method or block.
4) Character values has to be enclosed with single quote. Example: 'A'.
5) Assignement is done at the RHS, RHS operation is evaluated and stored in LHS as result.

Example: Program to print all the natural number between 1 to 10.

class NaturalNumber
{
    public static void main(String[] args)
    { for(int i=1;i<10;i++)
        {
            System.out.println("value of i is " +i);
        }
    }
}

Output: value of i is
1 value of i is 2
value of i is 3 value
of i is 4 value of i is
5 value of i is 6
value of i is 7 value
of i is 8 value of i is
9

Example: Program to print all the Odd number between 1 to 10.

class OddNumber
{
    public static void main(String[] args)
    { for(int i=1;i<10;i=i+2)
        {
            System.out.println("odd number are " +i);

```
        }
    }
}
```

Output:

odd number are 1 odd
number are 3 odd number
are 5 odd number are 7 odd
number are 9 **Example:**
Program to print all the
Even number between 1
to 10. class EvenNumber

```
{
    public static void main(String[] args)
    { for(int i=2;i<10;i=i+2)
        {
            System.out.println("Even number are " +i);
        }
    }
}
```

Output:

Even number are 2
Even number are 4
Even number are 6
Even number are 8

Example: Program to print multiples of 3 till 10.

```
class Multipleof3
{
    public static void main(String[] args)
    { for(int i=3;i<10;i=i*3)
        {
            System.out.println(i);
        }
    }
}
```

Output:

3
9

Example: Print the word fizz if number is completely divisible by 3 between the range of number 1 to 10. class Fizz

```
{
    public static void main(String[] args)
    {
        for(int i=1;i<=10;i++)
        {
```

```
                if(i%3==0)
                {
                    System.out.println("fizz=" +i);
                }
            }
        }
}
```

Output: fizz=3
fizz=6 fizz=9

_____ **Print Fizz if the number is completely divisible by 3, and print buzz if divisible by 5 and print FizzBuzz if the number is divisible bu both 3 and 5 between the range of 1 till 20.**

```
class FizzBuzz
{// first print the FizzBuzz condition because I%3==0 && I%5==0 may skip the 2nd condition.
    public static void main(String[] args)
    {
        for(int i=1;i<=20;i++)
        {
            if(i%3==0 && i%5==0)
            {
                System.out.println("FizzBuzz" +i);
            }
            else if(i%3==0)
            {
                System.out.println("Fizz" +i);
            }
            else if(i%5==0)
            {
                System.out.println("Buzz" +i);
            }
        }
    }
}
```

Output:

```
Fizz3
Buzz5
Fizz6
Fizz9
Buzz10
Fizz12
FizzBuzz15
Fizz18 Buzz20
```

Example:

```
class Simple17
{
    public static void main(String[] args)
    {
        for(int i=1;i<=5;i++)//Outer Loop
        {
            for(int j=1;j<=i;j++)//Inner Loop
            {
                System.out.print("*");//Printing *
            }
            System.out.println("");//Number of rows
        }
    }
```

```
}
```

Output:

```
*
**
***
****
*****
```

Note:

1) Outer For loop should concentrate on “Number of Rows”. 2) Inner For loop should concentrate on “Printing *”.

Execution:

Step 1: when i=1, is 1<=5? true then Goto Inner loop when j<=i, i.e., 1<=1? true print * now 2<=1? false then Goto Outer loop

Step 2: when i=2, is 2<=5? true then Goto Inner loop when j<=i, i.e., 1<=2? true and 2<=2? true print * * now 3<=2? false then Goto Outer loop

Step 3: when i=3, is 3<=5? true then Goto Inner loop when j<=i, i.e., 1<=3? true and 2<=3? true 3<=3?true print * * * now 4<=3? false then Goto Outer loop

Step 4: when i=4, is 4<=5? true then Goto Inner loop when j<=i, i.e., 1<=4? true and 2<=4? true 3<=4?true
4<=4? true print * * * * now 5<=4? false
then Goto Outer loop

Step 5: when i=5, is 5<=5? true then Goto Inner loop when j<=i, i.e., 1<=5? true and 2<=5? true 3<=5?true
4<=5? true 5<=5 true print * * * * *
Exit

Example:

```
class Simple18
{
    public static void main(String[] args)
    {
        int a=1; //2 //3 //4 for(int
        i=1;i<5;i++)
        { for(int j=1;j<=i;j++)
            {
                System.out.print(a ); a++;
            }
            System.out.println();
        }
    }
}
```

Output:

```
1
```

```
23
456
78910
```

_____ **Variable ch declared outside for loop** class

```
simple19
{
    public static void main(String[] args)
    {
        char ch='A';
        for(int i=1;i<5;i++)
        { for(int j=1;j<=i;j++)
            {
                System.out.print(ch);
                ch++;
            }
            System.out.println();
        }
    }
}
```

Output:

```
A
BC
DEF
GHIJ
```

Example: Variable ch declared inside for loop class

```
Simple20
{
    public static void main(String[] args)
    { for(int i=1;i<=5;i++)
        {
            char ch='A';
        //Character will be referehed, and will always start from A for(int
            j=1;j<=i;j++)
            {
                System.out.print(ch); ch++;
            }
            System.out.println();
        }
    }
}
```

Output:

```
A
AB
ABC
ABCD
```

ABCDE

Variable ch used as Condition in for loop class

```
Simple21
{
    public static void main(String[] args)
    {
        for(char ch1='A';ch1<='E';ch1++)
        {
            for(char ch2='A';ch2<=ch1;ch2++)
            {
                System.out.print(ch1);
            }
                System.out.println();
        }
    }
}
```

Output:

```
A
BB
CCC
DDDD
EEEEE
```

Example: class

```
Simple22
{
    public static void main(String[] args)
    { for(int i=1;i<=5;i++)
        { for(int j=1;j<=i;j++)
            {
                System.out.print((i+j)%2);
            }
                System.out.println();
        }
    }
}
```

Output:

```
0
10
010
1010
01010
```

Note:

i=1;j=1=>System.out.print((1+1)/2) i=2;j=1->2=>System.out.print((2+1)/2) class Simple23

```
{
```

```
    public static void main(String[] args)
    { for(int i=1;i<=5;i++)
        { for(int j=0;j<i;j++)
            {
                System.out.print((i+j)%2);
            }
            System.out.println();
        }
    }
}
```

Output:

```
1
01
101
0101
10101
```

Example:

1) int a, b, c; // declares three ints, a, b, and c.
2) int d = 3, e, f = 5; // declares three more ints, initializing // d and f.
3) byte z = 22; // initializes z.
4) double pi = 3.14159; // declares an approximation of pi. 5) char x = 'x'; // the variable x has the value 'x'.

Constants

In Java, you use the keyword final to denote a constant.

Example:

```
public class Constants
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5; double
        paperHeight = 11;
        System.out.println("Paper size in centimeters: " + paperWidth * CM_PER_INCH + " by " +
                                paperHeight * CM_PER_INCH);
    }
}
```

Note:

1) The keyword final indicates that you can assign to the variable once, and then its value is set once and for all. It is customary to name constants in all uppercase.

2) It is probably more common in Java to create a constant so it's available to multiple methods inside a single class. These are usually called class constants. 3) Set up a class constant with the keywords static final.

A class constant:

```
public class Constants2
{
    public static final double CM_PER_INCH = 2.54;
    public static void main(String[] args)
    {
        double paperWidth = 8.5; double
        paperHeight = 11;
        System.out.println("Paper size in centimeters: " + paperWidth * CM_PER_INCH + " by " +
                                    paperHeight * CM_PER_INCH);
    }
}
```

Note:

1) The definition of the class constant appears outside the main method.
2) Thus, the constant can also be used in other methods of the same class.
3) Furthermore, if the constant is declared, as in the example , public methods of other classes can also use it—in our example, as Constants2.CM_PER_INCH.

Dynamic Initialization

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

Example: program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

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

Note:

1) Here, three local variables—a, b, and c—are declared.
2) The first two, a and b, are initialized by constants.
3) However, c is initialized dynamically to the length of the hypotenuse.
4) The program uses another of Java's built-in methods, sqrt( ), which is a member of the Math class, to compute the square root of its argument.

5) The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

Scope and Lifetime of Variables

1) Java allows variables to be declared within any block.
2) A block is begun with an opening curly brace and ended by a closing curly brace.
3) A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
4) A scope determines what objects are visible to other parts of your program.
5) It also determines the lifetime of those objects.
6) In Java, the two major scopes are those defined by a class and those defined by a method.

Rule:

1) Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.

2) Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

3) Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested. 4) Each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.

5) This means that objects declared in the outer scope will be visible to code within the inner scope.

6) The reverse is not true. Objects declared within the inner scope will not be visible outside it.

Example: class

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

Note:

1) Variables are created when their scope is entered, and destroyed when their scope is left.

2) This means that a variable will not hold its value once it has gone out of scope.

3) Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left.

4) If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

Example: Demonstrate lifetime of a variable.

```
class LifeTime { public static void main(String
args[])
    {
        int x;
        for(x = 0; x < 3; x++)
        {
            int y = -1; // y is initialized each time block is entered
            System.out.println("y is: " + y); // this always prints -1
            y = 100;
            System.out.println("y is now: " + y);
        }
    }
```

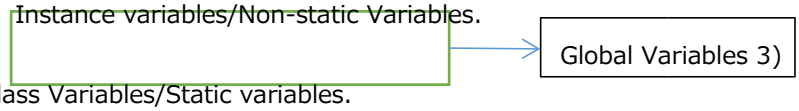
```
}
```

Outputy is: -1 y is now: 100 y is: -1 y is now: 100 y is: -1 y is now: 100

Kinds of variables in Java:

There are three kindsof variables in java:

1) Local variables.
2) Instance variables/Non-static Variables.
3) Class Variables/Static variables.



1) Local Variables:

a) Variables which are declared inside a method or block are called as Local Variables.
b) Local Variables scope/Visibility is only inside a method in which it is declared, i.e., Local Variables cannot be accessed outside the method in which it is created/declared.
c) Access modifiers cannot be used for local variables.
d) Local var are implemented at stack level internally & executed when method enters the stack.
e) Local variables cannot be categorized into static or non-static variables.

2)Instance Variables(Non-static Variables):

a) Instance variables are declared in a class, but outside a method, constructor or any block.
b) Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed
c) Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
d) Instance variables can be declared in class level before or after use.
e) Access modifiers can be given for instance variables.
f) The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level)
g) Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
h) Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . ObjectReference.VariableName.

3)Class Variables(Static Variables):

a) Static variables are declared with thestatic keyword in a class, but outside a method, constructor or a block.
b) There would only be one copy of each class variable per class, regardless of how many objects are created from it.

c) Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
d) Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
e) Static variables are created when the program starts and destroyed when the program stops.
f) Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class
g) Default values are same as instance variables.
h) Static variables can be accessed by calling with the class name ClassName.VariableName.
i) When declaring class variables as public static final, then variables names (constants) are all in upper case.

3. OPERATORS

1) Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

Operator	Result
+	Addition
-	– Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

Note:

1)The operands of the arithmetic operators must be of a numeric type.

2) You cannot use them on Boolean types, but you can use them on char types.

Example: Demonstrate the basic arithmetic operators. class BasicMath

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

```
        System.out.println("Integer Arithmetic"); int
        a = 1 + 1; int b = a * 3; int c = b / 4; int d
        = c - a; int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);
        System.out.println("¥nFloating Point Arithmetic");
        double da = 1 + 1; double db = da * 3; double dc =
        db / 4; double dd = dc - a; double de = -dd;
        System.out.println("da = " + da);
        System.out.println("db = " + db);
        System.out.println("dc = " + dc);
        System.out.println("dd = " + dd);
        System.out.println("de = " + de);
    }
}
```

Output:

```
Integer Arithmetic
a = 2 b = 6
c = 1 d = -1
e = 1
Floating Point Arithmetic
da = 2.0 db =
6.0
```

---

2) Modulus Operator:

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Example:

```
// Demonstrate the % operator. class
Modulus
{
    public static void main(String args[])
    {
        int x = 42; double y =
        42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```

Output:

```
x mod 10 = 2 y mod
10 = 2.25
```

---

3) Arithmetic Compound Assignment Operators:

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

Example: a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

This version uses the += compound assignment operator. Both statements perform the same action,they increase the value of a by 4.

Example:

a = a % 2;

which can be expressed as

a %= 2;

In this case, the %= obtains the remainder of a/2 and puts that result back into a.

There are compound assignment operators for all of the arithmetic, binary operators.

Thus, any statement of the form var = var op expression; can be rewritten as var op= expression;

Example: Demonstrate several assignment operators.

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

        int a = 1; int b = 2; int c = 3;
        a += 5; b *= 4; c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Output:

a = 6 b = 8

c = 3

---

4) Increment and Decrement:

The increment operator increases its operand by one.

The decrement operator decreases its operand by one.

Example:

a) x = x + 1; can be rewritten like this by use of the increment operator: b) x++;

Similarly, this statement: c) x = x - 1;

is equivalent to

x--;

Note:

In prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.

x = 42; y = ++x;

In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y. Thus, the line y = ++x; is the equivalent of these two statements:

x = x +1; y = x;

In postfix form, the previous value is obtained for use in the expression, and then the operand is modified. x = 42; y = x++;

In this case, value of x is obtained before the increment operator is executed, so the value of y is 42. Of course, in both cases x is set to 43. Here, the line y = x++; is the equivalent of these two statements: y = x; x = x + 1;

Example: Demonstrate ++.

```
class IncDec
{
    public static void main(String args[])
    {
        int a = 1; int b =
        2;
        int c; int d; c
        = ++b;
        d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

Output:

a = 2 b = 3 c = 4 d = 1

---

5)Bitwise Operators:

Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

----------5) a. The Bitwise Logical Operators:

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation.

A	B	A|B	A&B	A^B	B~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand.

Example: the number 42, which has the following bit pattern:

00101010 becomes 11010101 after

the NOT operator is applied.

The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A 'O' is produced in all other cases.

Example:

00101010 42
& 00001111 15
00001010 10

Example: Demonstrate the bitwise logical operators.

```
class BitLogic
{
    public static void main(String args[])
    {
String binary[] ={ "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001",
"1010", "1011", "1100", "1101", "1110", "1111" }; int a = 3; // 0
        + 2 + 1 or 0011 in binary int b = 6; // 4 + 2 + 0 or
        0110 in binary int c = a | b; int d = a & b; int e = a ^ b;
        int f = (~a & b) | (a & ~b); int g = ~a & 0x0f;
        System.out.println(" a = " + binary[a]);
        System.out.println(" b = " + binary[b]);
        System.out.println(" a|b = " + binary[c]);
        System.out.println(" a&b = " + binary[d]);
        System.out.println(" a^b = " + binary[e]);
        System.out.println("~a&b|a&~b = " + binary[f]);
        System.out.println(" ~a = " + binary[g]);
    }
}
```

Output: a = 0011 b =
0110 a|b = 0111 a&b =
0010 a^b = 0101
~a&b|a&~b = 0101
~a = 1100

5) b. Left Shift:

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times.

Syntax:

value << num

Here, num specifies the number of positions to left-shift the value in value.
That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num.
For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63.

Example: Left shifting a byte value.

```
class ByteShift
{
    public static void main(String args[])
    {
        byte a = 64, b;
```

```
        int i;
        i = a << 2; b = (byte)
        (a << 2);
        System.out.println("Original value of a: " + a);
        System.out.println("i and b: " + i + " " + b);
    }
}
```

Output:

Original value of a: 64 i and
b: 256 0

5) b. Right Shift

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.

Syntax:

value >> num

Here, num specifies the number of positions to right-shift the value in value.
That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num.

The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8:

```
int a = 32; a = a >> 2; // a now
contains 8
```

When a value has bits that are "shifted off," those bits are lost.
For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in a being set to 8.

```
int a = 35;
a = a >> 2; // a still contains 8
```

----------**6) Relational Operators:**

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering.

The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to

<=	Less than or equal to

7) Boolean Logical Operators:

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The following table shows the effect of each logical operation:

A	B	A|B	A&B	A^B	!A
False	False	False	False	False	True
True	True	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Example: Here is a program that is almost the same as the BitLogic example shown earlier, but it operates on boolean logical values instead of binary bits:

```
class BoolLogic
{
    public static void main(String args[])
    {
        boolean a = true; boolean b =
        false; boolean c = a | b; boolean
        d = a & b; boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
```

```
        System.out.println(" a|b = " + c);
        System.out.println(" a&b = " + d);
        System.out.println(" a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println(" !a = " + g);
    }
}
```

After running this program, you will see that the same logical rules apply to boolean values as they did to bits.

Output: the string representation of a Java boolean value is one of the literal values true or false:
a = true b = false a|b
= true a&b = false
a^b = true a&b|a&!b
= true !a = false

---

8) Assignment Operator:

The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language.

Syntax: var = expression;

Here, the type of var must be compatible with the type of expression.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments.

Example:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement.

This works because the = is an operator that yields the value of the right-hand expression.

Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x.

Using a "chain of assignment" is an easy way to set a group of variables to a common value.

---

9) ? Operator:

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements.

This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered.

Syntax: expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value.

If expression1 is rue, then expression2 is evaluated; otherwise, expression3 is evaluated.

The result of the ? operation is that of the expression evaluated.

Both expression2 and expression3 are required to return the same type, which can't be void.

Example: Demonstrate ?.

class Ternary

```
{
    public static void main(String args[])
    { int i, k; i = 10; k = i < 0 ? -i : i; // get
        absolute value of i
        System.out.print("Absolute value of ");
        System.out.println(i + " is " + k); i = -10;
        k = i < 0 ? -i : i; // get absolute value of i
        System.out.print("Absolute value of ");
        System.out.println(i + " is " + k);
    }
}
```

Output:

Absolute value of 10 is 10

Absolute value of -10 is 10

---

10) Operator Precedence:

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

4. CONTROL STATEMENTS

Java's program control statements can be put into the following categories:

1) Selection.
2) Iteration.
3) jump.

1) Selection statements: allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
2) Iteration statements: enable program execution to repeat one or more statements (that is, iteration statements form loops).
3) Jump statements: allow your program to execute in a nonlinear fashion.

Selection Statements

Java supports two selection statements:
A) If B) switch.
These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

A) I. If:

The if statement is Java's conditional branch statement. It can be used to route program execution
Syntax:

if (condition) statement1;
else statement2; Here:

each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block).

The condition is any expression that returns a boolean value. The else clause is optional.

Working:

If the condition is true, then statement1 is executed.
Otherwise, statement2 (if it exists) is executed.
In no case will both statements be executed.

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0; //Here, if a is less than b, then a is set to zero. Otherwise, b is set to
    zero
```

A) II. Nested ifs:

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

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

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

A) III. if-else-if Ladder:

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

Syntax:

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

The if statements are executed from the top down.

As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.

If none of the conditions is true, then the final else statement will be executed.

The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed.

If there is no final else and all other conditions are false, then no action will take place.

Example: Demonstrate if-else-if statements.

```
class IfElse
{
    public static void main(String args[])
    {
        int month = 4; // April String
        season;
        if(month == 12 || month == 1 || month == 2)
        season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
        season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
        season = "Summer"; else if(month == 9 || month
        == 10 || month == 11)
        season = "Autumn"; else
        season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
}
```

Output:

April is in the Spring.

B) I. switch

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.

Syntax: switch (expression)

```
{
    case value1:
```

```
            // statement sequence
        break; case value2:
            // statement sequence break;
        ... case valueN:
            // statement sequence
        break; default:
            // default statement sequence
    }
```

Rules:

1) The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression.

2) Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

Working:

1) The value of the expression is compared with each of the literal values in the case statements.
2) If a match is found, the code sequence following that case statement is executed. 3) If none of the constants matches the value of the expression, then the default statement is executed.
4) However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

Note:

The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. This has the effect of "jumping out" of the switch.

Example:A simple example of the switch.

```
class SampleSwitch
{
    public static void main(String args[])
    {
        for(int i=0; i<6; i++)
        switch(i)
        {
        case 0:
            System.out.println("i is zero.");
        break; case 1:
            System.out.println("i is one.");
        break; case 2:
            System.out.println("i is two.");
        break; case 3:
            System.out.println("i is three.");
        break; default:
            System.out.println("i is greater than 3.");
        }
    }
}
```

Output: i is zero. i is
one. i is two. i is three.
i is greater than 3. i is
greater than 3.

B) II. Nested switch Statements:

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch.

Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.

Syntax: switch(count)

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

Here, the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch.

The count variable is only compared with the list of cases at the outer level. If count is 1, then target is compared with the inner list cases.

Summary:

there are three important features of the switch statement to note:

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs.

Jump Statements

Java supports three jump statements: A) break.
B) continue.
C) return.
These statements transfer control to another part of your program.

A) break:

In Java, the break statement has three uses.

1) It terminates a statement sequence in a switch statement.
2) It can be used to exit a loop.
3) It can be used as a “civilized” form of goto.

2) Using break to Exit a Loop:

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

Example: Using break to exit a loop.

```
class BreakLoop
{
    public static void main(String args[])
    {
        for(int i=0; i<100; i++)
        {
        if(i == 10) break; // terminate loop if i is
            10
        System.out.println("i: " + i);
        }
    System.out.println("Loop complete.");
    }
}
```

Output:

```
i: 0 i: 1 i: 2 i: 3 i: 4
i: 5 i: 6 i: 7 i: 8 i: 9
Loop complete.
```

Working:

Although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when i equals 10.

The break statement can be used with any of Java's loops, including intentionally infinite loops.

Example: Using break to exit a while loop. class

```
BreakLoop2
{
    public static void main(String args[])
    {
        int i = 0;
        while(i < 100)
         {
            if(i == 10) break; // terminate loop if
            i is 10 System.out.println("i: " + i);
            i++;
        }
        System.out.println("Loop complete.");
    }
}
```

Example: Using break with nested loops. class

```
BreakLoop3
```

```
{
    public static void main(String args[])
    {
        for(int i=0; i<3; i++)
        {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++)
            {
                if(j == 10) break; // terminate loop
                if j is 10 System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}
```

Output:

```
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

Points to remember:

First, more than one break statement may appear in a loop. However, be careful. Too many break statements have the tendency to destruct your code.

Second, the break that terminates a switch statement affects only that switch statement and not any enclosing loops.

3) Using break as a Form of Goto:

Syntax:

```
break label;
```

Example: Using break as a civilized form of goto. class Break

```
{
    public static void main(String args[])
    {
        boolean t = true; first:
        {
            second:
            { third:
                {
                    System.out.println("Before the break.");
                if(t)
                    break second; // break out of second block System.out.println("This
                    won't execute");
                }
                System.out.println("This won't execute");
```

```
            }
            System.out.println("This is after second block.");
        }
    }
}
```

Output:

Before the break.
This is after second block.

Example: Using break to exit from nested loops

```
class BreakLoop4
{
    public static void main(String args[])
    {
        outer: for(int i=0; i<3; i++)
        {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++)
            {
                if(j == 10) break outer; // exit both
                loops System.out.print(j + " ");
            }
            System.out.println("This will not print");
        }
        System.out.println("Loops complete.");
    }
}
```

Output:

Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

B) continue:

In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.

In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

For all three loops, any intermediate code is bypassed.

Example: Demonstrate continue. class

```
Continue
{
    public static void main(String args[])
     {
        for(int i=0; i<10; i++)
        {
```

```
                System.out.print(i + " "); if
                (i%2 == 0)
//This code uses the % operator to check if i is even. If it is, the loop continues without
                printing a newline. continue;
                System.out.println("");
            }
        }
}
```

Output:

```
0                                                                    18
2                                                                    26
4                                                                    45
6                                                                    48
8                                                                    77
```

Example: Using continue with a label. class

```
ContinueLabel
{
    public static void main(String args[])
    {
        outer: for (int i=0; i<10; i++)
        /*continue statement in this example terminates the loop counting j and continues with
        the next iteration of the loop counting I*/
        {
            for(int j=0; j<10; j++)
            { if(j > i)
                {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
        }
        System.out.println();
    }
}
```

Output:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
```

0 9 18 27 36 45 54 63 72 81

C) return:

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method.

Thus, the return statement immediately terminates the method in which it is executed.

Example: Demonstrate return.

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

Output:

```
Before the return.
```

Iteration Statements

Java's iteration statements are

1) for
2) while
3) do-while.

These statements create what we commonly call loops.

A loop repeatedly executes the same set of instructions until a termination condition is met.

1) while:

It repeats a statement or block while its controlling expression is true.

Syntax:

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

Here,

The condition can be any Boolean expression.

The body of the loop will be executed as long as the conditional expression is true.

When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Example: while loop that counts down from 10, printing exactly ten lines of "tick":

```
class While
{
    public static void main(String args[])
    {
        int n = 10;
        while(n > 0)
        {
            System.out.println("tick " + n); n--;
        }
    }
}
```

Output: When you run this program, it will "tick" ten times:

tick 10 tick 9 tick 8 tick 7 tick 6 tick 5 tick 4 tick 3 tick 2 tick 1

Note:Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

---

2) do-while:

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Syntax:

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

Note:

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.

If this expression is true, the loop will repeat. Otherwise, the loop terminates.

As with all of Java's loops, condition must be a Boolean expression.

Example:

```
class
DoWhile
{
    public static void main(String args[])
    {
        int n = 10; do
        {
            System.out.println("tick " + n); n--;
        }
          while(n > 0);
```

```
    }
}
/*The loop in the preceding program, while technically correct, can be written more efficiently as
follows:*/
do
{
    System.out.println("tick " + n);
}
 while(--n > 0);
```

Usage: The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

Example:Using a do-while to process a menu selection class

```
Menu
{
    public static void main(String args[]) throws java.io.IOException
    {
        char choice;
        do
        {
            System.out.println("Help on:");
            System.out.println(" 1. if");
            System.out.println(" 2. switch");
            System.out.println(" 3. while");
            System.out.println(" 4. do-while");
            System.out.println(" 5. for¥n"); System.out.println("Choose
            one:");
            choice = (char) System.in.read();
        }
        while( choice < '1' || choice > '5');0
        System.out.println("¥n");
        switch(choice)
        {
            case '1':
                System.out.println("The if:¥n");
                System.out.println("if(condition) statement;");
                System.out.println("else statement;");
            break; case '2':
                System.out.println("The switch:¥n");
                System.out.println("switch(expression) {");
                System.out.println(" case constant:");
                System.out.println(" statement sequence");
                System.out.println(" break;");
                System.out.println(" // ...");
```

```
                System.out.println("}");
            break; case '3':
                System.out.println("The while:¥n");
                System.out.println("while(condition)     statement;");
            break; case '4':
                System.out.println("The do-while:¥n");
                System.out.println("do {");
                System.out.println(" statement;");
                System.out.println("} while (condition);"); break;
            case '5':
                System.out.println("The for:¥n");
                System.out.print("for(init; condition; iteration)");
                System.out.println(" statement;"); break;
        }
    }
}
```

Output:

```
Help on:
1. if
2. switch
3. while
4. do-while 5. for
Choose one:
4
The do-while: do {
statement; } while
(condition);
```

---

3) for:

Beginning with JDK 5, there are two forms of the for loop. A) "for" form.
B) "for-each" form.

Syntax:

```
for(initialization; condition; iteration)
{
}
```

3. A) for loop:

1) When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. 2) Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control

variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
3) Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass.
This process repeats until the controlling expression is false.

Example: Demonstrate the for loop.

```
class ForTick
{
    public static void main(String args[])
    {
        int n;
        for(n=10; n>0; n--)
        System.out.println("tick " + n);
    }

}
```

3. B) for-Each:

A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.
The advantage of this approach is that no new keyword is required, and no preexisting code is broken. Syntax:

```
for(type itr-var : collection) statement-block
```

Here, type specifies the type itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
The collection being cycled through is specified by collection.
With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var.
The loop repeats until all elements in the collection have been obtained.
Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection.
Thus, when iterating over arrays, type must be compatible with the base type of the array.

NOTE:

1. Enhanced for loop was designed in JDK 1.5.
2. To fetch the element stored in array and collections without any condition, we can use extended for loop.
3. Enhanced for loop is also called as ”for-each” loop because each and every element will be fetched without fail.
4. Enhanced for loop can be used only for arrays and collections.
5. If you want to perform a common operation on all the elements stored in a array or collection then best approach is using enhanced for loop Syntax:

```
for(arraytype varaible : Arrayreference)
{    }
```

Example: Use a for-each style for loop. class

```
ForEach
{
    public static void main(String args[])
    {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7};
        int sum = 0;
        // use for-each style for to display and sum the values for(int
        x : nums)
        {
            System.out.println("Value is: " + x); sum
            += x;
        }
        System.out.println("Summation: " + sum);
    }
}
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Summation: 28
```

Example: Use break with a for-each style for. class

```
ForEach2
{
    public static void main(String args[])
     {
        int sum = 0; int nums[] = { 1, 2, 3, 4, 5,
        6, 7, 8, 9, 10 }; // use for to display and
        sum the values
        for(int x : nums)
        {
            System.out.println("Value is: " + x); sum +=
        x; if(x == 5) break; // stop the loop when 5 is
        obtained
        }
    System.out.println("Summation of first 5 elements: " + sum);
    }
}
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
```

Value is: 4
Value is: 5
Summation of first 5 elements: 15

3. C) Nested loops:

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

Example: Loops may be nested.

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

Output:

```
..........
.........
........
.......
......
.....
....
...
.. .
```

---

6. METHODS

Introduction

Definition: A Java Method is a collection of statements that are grouped together to perform an operation and executed whenever called or invoked.

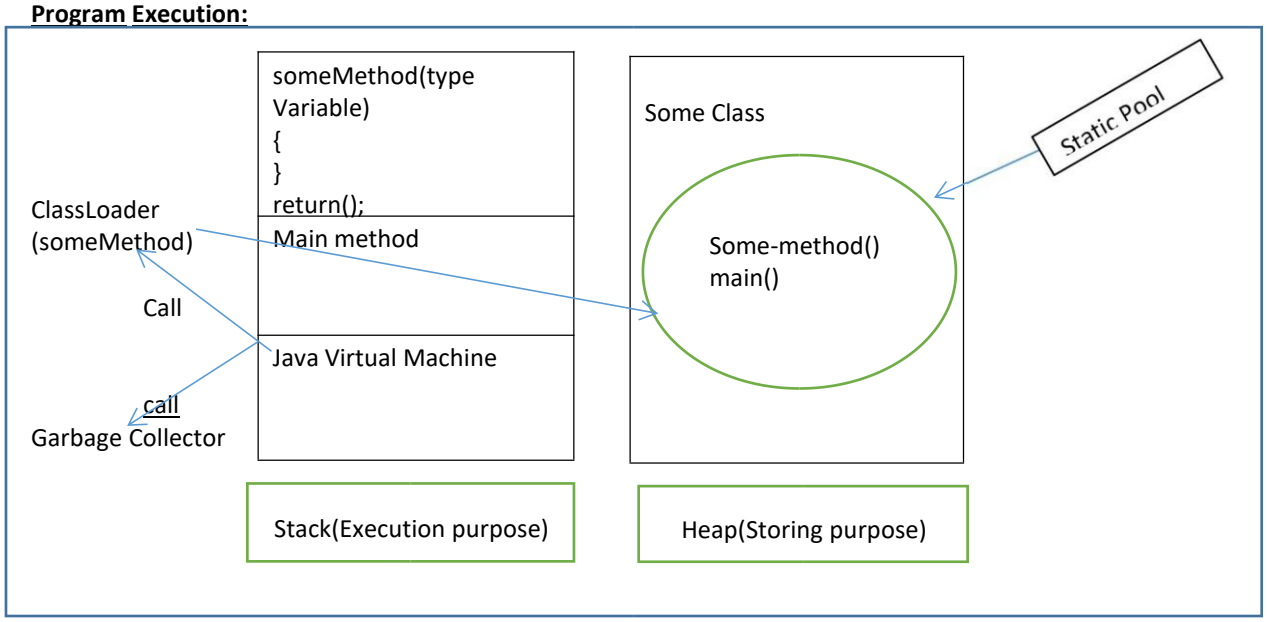
Why methods?
Re-Usability

Syntax:

```
Access-level modifier return-type method-name(arguments)
{
    Body of method
}
```

Access-level: public, protected, default, private.
Modifier: static or non-static.
Return-type: Primitive data-type or Derived data-type. Method-name:Is an identifier can be any valid identifier. arguments:inputs for a method.
Note: Return type and return value should match.

Program Execution:



Part-1:

The following process takes place when we execute a program: 1)
Program execution happens in RAM.

2) RAM memory is divided into 2 types,
   a) Stack:For execution purpose.
   b) Heap:For Storage purpose.
3) JVM will be loaded onto stack automatically.
4) JVM will call class loader (Built-in Library).
5) Class Loader will create a static pool in the heap memory and all the static members of the class will be loaded on to it.
6) Static pool name will be similar to class name.

Part-2:

1) JVM will check the availability of main method in the static pool, if its available, JVM will load the main method for execution purpose.
2) JVM will pass the control to main method so that main method can execute its statement.

Part-3:

1) From main method, some method is called by using static pool name i.e., Classname.someMethod(1);
2) someMethod() will be loaded onto the stack.
3) Control will pass to the someMethod(), so it can executes its statement.
4) someMethod() after its execution will return the control back to main method.

5) someMethod() will be erased from the stack.
6) Main method resume its execution and finally it will return the control back to JVM.
7) Main method will be erased from stack.
8) JVM will call Garbage Collector(Daemon Thread).
9) Garbage Collector will clear the contents from the HEAP memory. 10) JVM will exit from the stack.

Example:

```
public class Simple1
{
    static void square(int num)
    {
        int sq = num*num;
        System.out.println("Result is" +sq); return;
    }
    public static void main(String[] args)
    {
        Simple1.square(2);//Method invocation
    }
}
```

Output: 4

Example:

```
public class Simple2
{
    static void cube(int num)
    {
        int c=num*num*num;
        System.out.println("Result is: " +c);

    }
    public static void main(String[] args)
    {
        Simple2.cube(3);
    }
}
```

Output:

Result is: 27

Note:

1) void means return no value but return control.
2) If the method return-type is void, then that method cannot return any value after processing.
3) If any methods return-type is void, then developing return statement is not mandatory. 4) If any methods return-type is void and if the programmer has not developed to return statement then compiler will develop return statement automatically at compile time.

5) Developing methods with arguments is not mandatory i.e., we can develop methods without arguments.
6) The methods without arguments cannot receive any input at the time of method invocation.

Example: Interview Question

```
public class Simple5
{
    static test()// Return type required
    {
        System.out.println("Running test method()");
    }
    public static void main(String[] args)
    {
        Simple5.test();
    }
}
```

Output:

CTE
Return type required

Note:

Can we develop a method without return type?

No, we cannot develop any method without return type, return type is mandatory should be either a primitive data type or Derived Data-type or void, but arguments for a method is not mandatory

Example: Non-void method

```
public class Simple6
{
    static int square(int num)
    {
        int sq=num*num;
        return sq;
    }
    public static void main(String[] args)
    {
        int res=Simple6.square(9); System.out.println("Result
        is :" +res);
    }
}
```

Output:

Result is :81

Example: public class
Simple7
{ static int test()

```
    {
        return 90;
    }
    public static void main(String[] args)
    {
        int res=Simple7.test();
        System.out.println("Result is :" +res);
    }
}
```

Output: Result is
:90

Example:

```
public class Simple8
{
    static double test()
    {
        return 90.5;
    }
    public static void main(String[] args)
    {
        System.out.println(Simple8.test());
    }
}
```

Output:
90.5

Note:
In System.out.println we cannot call the method of type void, which cannot return value.
We can call only non-void method.

Example: Void type not allowed in “System.out.println” public

```
class Simple9
{
    static void test()
    {
        return;
    }
    public static void main(String[] args)
    {
        System.out.println(Simple9.test());// Void type not allowed here
    }
}
```

Output:
Erroneous tree type:
Note:

If the methods Return type is void, we cannot call it in "System.out.println" or having the "System.out.println" we can only call those methods which will return some value(non-void method).

When we opt for void method?

Whenever we are not expecting any return value from the method agter processing then we should develop those method with return type.

When we opt for non-void method?

Whenever we are expecting the return value from the method and based on that return value if we want to do further processing then we should choose non-void method.

Example: Given 2 integers return true(boolean) if sum of them is 30 or one of them is 30.

```
public class Simple10
{
    static boolean test(int n1,int n2)
    {
        if(n1+n2==30||n1==30||n2==30)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public static void main(String[] args)
    {
        System.out.println("Result is " +Simple10.test(10, 20));
        System.out.println("Result is " +Simple10.test(20, 20));
    }
}
```

Output:

```
Result is true
Result is false
```

Example: Given 2 integers return twice there sum if both are same otherwise return their sum.

```
public class Simple11
{
    static int twiceSum(int a,int b)
    { if(a==b)
        {
            return 2*(a+b);
        }
        else return (a+b);
    }
    public static void main(String[] args)
```

```
    {
        System.out.println("Result is :" +Simple11.twiceSum(10,20));
        System.out.println("Result is :" +Simple11.twiceSum(10,20));
        System.out.println("Result is :" +Simple11.twiceSum(10,10));
    }
}
```

Output:

```
Result is :30
Result is :30
Result is :40
```

Example: There are 2 monkeys, if both the monkeys are smiling then we are in trouble, if both the monkeys are not smiling then also we are in trouble, Return true if we are in trouble.

```
public class Simple12
{
    static boolean trouble(boolean a, boolean b)
    {
        if(a==true && b==true)
        {
            return true;
        }
        else if(a==false && b==false)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public static void main(String[] args)
    {
        System.out.println("Result is:" +Simple12.trouble(true, true));
        System.out.println("Result is:" +Simple12.trouble(false, false));
        System.out.println("Result is:" +Simple12.trouble(true, false));
        System.out.println("Result is:" +Simple12.trouble(false, true));
    }
}
```

Output:

```
Result is:true
Result is:true
Result is:false
Result is:false
```

6. CLASSES AND OBJECTS

Class

Class is a collection of fields and Behavior.
Fields are also called as Variables or attributes or Data Members.
Behavior is also called as Functions or Methods or Member Functions.

Syntax:

```
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
    }
}
```

The data, or variables, defined within a class are called instance variables.
The code is contained within methods.

Member of a class

The methods and variables defined within a class are called members of the class.
There are 2 types of members: a)
Static Members.
b) Non-Static Members.

a) Static Members:
   a) Any Members declared with the keyword Static is called as Static Members.
   b) Static Members include Static Variables, static Methods and Static Initialization Block(SIB).
   c) To access the Static members of a class we should use classname.staticmembername, because static members will be loaded onto static pool by class loader and the static pool name would be similar to class name.

b) Non-Static Members:

a) Any members declared without the keyword Static keyword is called as Non-Static Members.
b) Non-Static Members includes Non-static Variables, Non-static Methods and Non-static blocks or Instance Initialization Block(IIB).
c) To access the non-static members of a class we should "Create Object of a class" or we should "create instance of a class" or we should "Instantiate the class".
d) To access the Non-Static members of a class we should use
Class-name reference-variables = new Class-name(

Example: To access non-static Members

```
public class Simple1 { void
    test()
    {
        System.out.println("Running test method()");
    }
    public static void main(String[] args)
    {
        Simple1 s=new Simple1();
        s.test();
    }
}
```

Output:

Running test method()

Note:

1) New operator is used to create new address space or non-static pool somewhere in the heap Memory.
2) Class-name() will load all the non-static members of class onto the address space created by new operator.
3) Reference variables will store the address of the object so that we can refer the object.
4) Class-name is a reference variable type i.e., reference variable(rv) is a reference variable of Class.

Example:

```
public class Simple2
{
    void test()
    {
        System.out.println("Runnig test method");
    }
    public static void main(String[] args)
    {
        Simple2 s2=new Simple2();
        s2.test();
        Simple2 s22=new Simple2(); s22.test();
    }
}
```

Output:

Runnig test method
Runnig test method

Example: Non-static Members, need to create Reference variable

```
public class Simple3
{
    int a=90;//Non-static Variables
    void test()//Non-static Methods
    {
        System.out.println("Running Test method");
    }
    public static void main(String[] args)
    {
        Simple3 s3=new Simple3();
        s3.test();
        System.out.println("Value of a" +s3.a);//Access Global variable "a" } }
```

Output:

Running Test method
Value of a9

Example: Static Members, need not create reference variable, just access using Clasname.staticmember-name public

```
class Simple4
{
    static int a=90;//Static variable
    static void test()//static member
    {
        System.out.println("Runnig test method");
    }
    public static void main(String[] args)
    {
        System.out.println("Value of a is " +Simple4.a);
        Simple4.test();
    }
}
```

Output:

Value of a is 90
Running test method

Example: To access static members and non-static memers.

```
public class Simple5
{
    static int a=10; //static varibale int
    b=20; //non-static varibale
    static void test()//static method
```

```
    {
        System.out.println("Running test method");
    }
    void test2()//non-static method
    {
        System.out.println("Running test2 method");
    }
    public static void main(String[] args)
    {
        Simple5.test();//Directly access the static method
        System.out.println("Value of a is " +Simple5.a);//Directly access the static varibale

        Simple5 s5=new Simple5(); //first create new address space and create a reference
        s5.test2();     //reference varibale used to access the non-static method which is in heap
                        memory
        System.out.println("Value of b is " +s5.b);
    }
}
```

Output:

Running test method
Value of a is 10
Running test2 method
Value of b is 20

Note:

1) We can have Single copy of static members.
2) If you want to have only one copy of any member e should declare the member with keyword static.
3) If the requirements is fixed then we should go for static members.
4) Ex: addition() logic.

5)

1) Whenever we need to have multiple copy of any member we should go for non-static.
2) If the requirements is not fixed then we should go for non-static members 3)
   Ex: salary(), percentage()

Note:

1) Method arguments are also Local Variables.
2) We can call a method by supplying the value through variables.
3) Local variables can have same name across different methods.
4) Local variables are method members and created when method enter into stack and destroyed when method leaves the stack after execution.

Example: Method arguments

```
public class Simple6
{
    static void test(int a)//Methods woth arguments
    {
```

```
        System.out.println("from test:" +a);
    }
    public static void main(String[] args)
    {
        int a=90;
        Simple6.test(a);
    }
}
```

Output: from
test:90

Example: Program to perform swapping of 2 numbers stored in the variables using temp variable.

```
public class Simple7
{
    static void swap(int num1,int num2)
    {
        int temp=num1; num1=num2;
            num2=temp;
            System.out.println("number1: " +num1);
            System.out.println("number2: " +num2);
    }
    public static void main(String[] args)
    {
        int num1=10; int
        num2=20;
        System.out.println("number1:" +num1);
        System.out.println("number2:" +num2);
        Simple7.swap(num1, num2);
    }
}
```

Output:

number1:10 number2:20
number1: 20 number2:
10

Example: Program to perform swapping of 2 numbers stored in the variables without using temp variable.

```
public class Simple8
{
    static void swap(int a, int b)
    {
            a=a+b; b=a-b;
            a=a-b;
            System.out.println("a=" +a);
```

```
            System.out.println("b=" +b);
    }
    public static void main(String[] args)
    {
        int a=10; int
        b=20;
        System.out.println("Before Swapping");
        System.out.println("a=" +a);
        System.out.println("b=" +b);
        System.out.println("After Swapping");
        Simple8.swap(a, b);
    }
}
```

Output: Before
Swapping a=10 b=20
After Swapping a=20
b=10

Example: Program to perform swapping of 2 numbers stored in the variables without using temp variable.

```
public class Simple9          {
    static void swap(int a, int b){ a=(b-a)+(b=a);
        System.out.println("a=" +a);
        System.out.println("b=" +b);
    }
    public static void main(String[] args)
    {
        int a=10; int
        b=30;
        System.out.println("Before Swapping");
        System.out.println("a= " +a);
        System.out.println("b= " +b);
        System.out.println("Aftwr Swapping");
        Simple9.swap(a, b);
    }
}
```

Output: Before
Swapping a= 10
b= 30
Aftwr Swapping
a=30 b=10

Note:

1) **Global variables** will reside in the **heap memory,** it will be there till the end of program.
2) If its **STATIC** , will be in **static pool**(class Variable).

3) If its **NON-STATIC**, will be in **Object space**(Instance Variable).

Example: public class Simple10

```
{
    static int count=0;
    static void click()
    {
        System.out.println("Button clicked");
        Simple10.count++;
    }
    public static void main(String[] args)
    {
        System.out.println("Total clicks: " +Simple10.count);
        Simple10.click();
        Simple10.click();
        System.out.println("Total clicks: " +Simple10.count);
        Simple10.click();
        Simple10.click();
        System.out.println("Total clicks: " +Simple10.count);
    }
}
```

Output:

```
Total clicks: 0
Button clicked
Button clicked
Total clicks: 2
Button clicked
Button clicked Total
clicks: 4
```

Note:

1) By default, local variables value is not assigned. Programmer should assign it.
2) Before using any variables it should have some value.
3) Global variables will be assigned with default value.
4) Local variables will not be assigned with default value.
5) Global variables can b e used without initializing it explicitly.
6) Local variables should be initialized before usage.
7) Static means 1 copy, non-static means multiple copy. 8) Any kind of variables(static.non-static) can be FINAL.

Interview Questions Simple11

```
    public static void main(String[] args)
    {
        int a; a++;
        System.out.println("Done");
    }
}
```

Output:

variable a might not have been initialized

Example:

```
public class Simple12
{
    public static void main(String[] args) {
        {
            int a;
            System.out.println("Done");
        }
    }
}
```

Output: Done

Example:

```
public class Simple13
{
    public static void main(String[] args) { int a;
        System.out.println("a++");//a++ is treated as String varibale
    }
}
```

Output: a++

FINAL

1) Final is a keyword.
2) If you want to declare constant variable where-in value cannot be changed then those variable should be declared with the keyword “Final”.
3) Final variable value cannot be changed or overridden.
4) Both local and global variable can be Final.
5) Global Final variables should be initialized at the time of declaration itself. 6) Local Final variables can be declared once and initialized later.

Note:

1) Static members of the same class can be accessed directly.

2) Local variables will be given preference over Global Variables.

Simple1

```
final static double PI=3.14;
public static void main(String[] args)
{
    System.out.println("PI value" +Simple1.PI);
    Simple1.PI=9.36;//cannot assign a value to final varibale
    {
        System.out.println("PI vlaue" +Simple1.PI);
    }
}
}
```

Output:

cannot assign a value to final variable PI

Example:

```
public class Simple2
{
    final static double PI;//Global final should be initialized at declaration itself public
    static void main(String[] args)
    {
        System.out.println("Value of PI is" +Simple2.PI);
    }
}
```

Output: variable PI not initialized in the default constructor

Example: If local and Global variables have the same name, then we should use class-name to access the global varibale

```
public class Simple3
{
    static int a=10;//Global varibale
    public static void main(String[] args)
    {
        int a=20;//Local Varibale System.out.println("local
        a=" +a);
        System.out.println("Global a=" +Simple3.a);
    }
```

Output: local a=20
Global a=10

NOTE:

1) Static members of the same class can be access directly from Static context.

2) Static members of the same class can be accessed directly from non-static also.
3) Non-Static members can be accessed directly from another Non-Static context .
4) Non-Static members cannot be accessed directly from Static context, we should create object and we should use object reference.

```
Bank

int balance;//non-static varibale
void balance()//non-static method
{
    System.out.println("Current balance is " +balance);
}
void deposit(int damount)//
{
    balance=balance+damount;//non-static members can be accesed directly from another
non-static context
    System.out.println("you have deposited" +damount +"Rs"); balance();
}
void withdraw(int wamount)//non-static method
{
    if(wamount<=balance)
    {
        balance=balance-wamount;
        System.out.println("you have withdrawn" +wamount+ "Rs");
    }
    else
    {
        System.out.println("insufficient balance");
    }
    balance();
}
public static void main(String[] args)
{
    System.out.println("welcome to HDFC bank"); Bank
    b=new Bank();
    b.balance();
    System.out.println("Transaction details"); b.deposit(10000);
    System.out.println("Transaction details"); b.withdraw(10000);
    System.out.println("Transaction details"); b.withdraw(10000);
}
}
```

Output:

```
welcome to HDFC bank
Current balance is :0 Transaction
details you have deposited
:10000Rs
```

Current balance is :10000
Transaction details you have
withdrawn :10000Rs
Current balance is :0 Transaction
details insufficient balance :
Current balance is :0

7. METHOD OVERLOADING

Definition:

Developing multiple methods with the same name, but variations in the arguments list. Variations in the arguments list can be:1)
Number of arguments.

2) Types of arguments.

3) Position/order of arguments

Whenever we want to perform the common operation or task with the variations in the inputs, we should choose Method Overloading.

Example:

If you want to write th eprogrsm to perform addition of 2 numbers and 3 numbers, then we should develop 2 method with 1 to receive 2 inputs and 1 more to receive 3 inputs. Instead of developing this methods with different names, we can develop with same name "add".

Example:

If you are developing an application and you are giving 2 options for the user name and password and another with the mobile numbers and password. Here also we should develop 2 methods with the variation in the inputs instead of developing this methods with different name the operation is common, we can develop it with the same name.

Note: Through Method Overloading:-

1) We can achieve consistency in the method names, which are developed for common purpose.
2) It is easy to remember method name.
3) We can achieve efficiency In the program readability.
4) We can achieve compile-time Polymorphism through Overloading.
5) While Overloading method name should be same with the variations in the arguments list, other parts of method does not matter, i.e., Return type, modifiers, access level can be different. 6) We can overload static and non-static methods.

Interview Questions:

1) Can we overload main method?

Yes, we can but the program execution starts with main method having arguments as (String[] args)Other versions of the main program should be invoked by the programmer explicitly. 2) Can we overload Non-static method?

Yes.

3) Can the return type differ while overloading? Yes.

Example:

```
public class Simple1
{
    void greet()
    {
        System.out.println("Welcome to Home");
    }
    void greeting(String name)
    {
```

```
        System.out.println("Welcome to Home: " +name);
    }
    public static void main(String[] args)
    {
        Simple1 s=new Simple1();
        s.greet();
        s.greeting("JSM");
    }
}
```

Output:

Welcome to Home Welcome to
Home: JSM

Example:

```
public class Calculate
{
    static void area(int side)
    {
        int a=side*side;
        System.out.println("Area of square is" +a);
    }
   static void area(int length, int breadth)
    {
        int a=length*breadth;
        System.out.println("Area of rectangle is" +a);
    }
    static void area(double radius)
    {
        double d=3.14*radius*radius; System.out.println("Area
        of circle is" +d);
    }
    public static void main(String[] args)
    {
        area(2); area(2,3);
        area(3.0);
    }
}
```

Output:

Area of square is4
Area of rectangle is6 Area of circle
is28.2599999999999

Example:

```
public class
Simple2
{
    static void show(int a)
```

```
    {
        System.out.println("input recieved is: " +a);
    }
        void show(double d)
    {
        System.out.println("input received is: " +d);
    }
    public static void main(String[] args)
    {
        show(10); Simple2 s2=new
        Simple2();
        s2.show(20.5);
    }
}
```

Output:

input recieved is: 10 input
received is: 20.5 **Example:**

```
public class Simple3
{
    static void test(int a)
    {
        System.out.println("Running test(int a)");
    }
    static void test(int b)
    {
        System.out.println("Running test(int a)");
    }
    public static void main(String[] args) {
        { test(10);
        }
    }
}
```

Output:

method test(int) is already defined in class

Example:

```
public class OverloadingMain
{
    public static void main(int a)
    {
        System.out.println("Main(int a)");
    }
    public static void main(double a)
    {
```

```
        System.out.println("Main(double a)");
    }
    public static void main(String[] args) //Oroginal Main
    {
        main(10);
        main(10.5);
    }
}
```

Output:

Main(int a)
Main(double a)

9. INHERITANCE

Definition:

Getting the features or properties of one class from another class is called as Inheritance. -> The class from which other class acquires the features is called as Super class or Parent class or Base class.

-> The class which acquires the features or properties is called as Sub class or Child class or Derived Class.

Through Inheritance we can achieve, 1. Extensibility.
2. Code Optimization.
3. Code Re-usability.
4. Code Maintainability.

To Achieve inheritance between classes, we should use **Extends** keyword.

There are 4 types of Inheritance, 1. Single Level inheritance.
2. Multilevel inheritance.
3. Hybrid Inheritance.
4. Multiple inheritance.

1. Single Level inheritance: One class inheriting from only one Super class.
2. Multilevel inheritance: One class inheriting from another Sub class.
3. Hybrid Inheritance: Two or more class inheriting from common Super class.
4. Multiple inheritance: One class inheriting from multiple immediate Super class.

Note: Multiple inheritance is not possible is java through Classes instead happens through Interfaces

Note: Only non-static members of a class will be involved in Inheritance.

Example: Single Level inheritance:

```
package inheritance; class A
{
    int a=10;
    void test1()
    {
        System.out.println("Running test1");
```

```
    }
}
class B extends A
{
    double d=10.5;
    void demo1()
    {
        System.out.println("Running demo2");
    }
}
class SingleInheritance
{
    public static void main(String[] args)
    {

    }

}
```

Example: Multilevel Inheritance:

```
package corejava; class
Sample1
{
    void test1()
    {
            System.out.println("Running Test1");
    }
}
class Sample2 extends Sample1
{
    void test2()
    {
            System.out.println("Running Test2");
    }
}
class Sample3 extends Sample2
{
    void test3()
    {
            System.out.println("Running Test3");
    }
}
class MultiLevelInheritence
{
    public static void main(String[] args)
    {
```

```
        System.out.println("Main Starts"); Sample3
        rv1=new Sample3();
        rv1.test1(); rv1.test2();
        rv1.test3();
        System.out.println();
        Sample2 rv2=new Sample2();
        rv2.test1(); rv2.test2();
        //rv2.test3();
        System.out.println("Main Ends");
        }
}
```

Output:

```
Main Starts
Running Test1
Running Test2
Running Test3

Running Test1
Running Test2
Main Ends
```

super keyword

Definition: super keyword is used to access super class non-static members in case of inheritance between classes.

1) Has Effective use in method overriding than inheritance.
2) Due to method overriding super class implementation will be masked in the sub class.
3) To get the original/masked implementation in sub class we can use super keyword.
4) To avoid overrriding use Final keyword.
5) Declare class as Final then no method can be overridden.

Super Calling Statement:

1) Super Calling Statement is used to call Super Class Constructor in case of inheritance between Classes.
2) Super Calling Statement will call immediate Super constructor in case of inheritance.
3) Through Super Calling Statement we can achieve Constructor Chaining.
4) Constructor Chaining means Sub class Constructor calling its immediate Super class Constructor.

Rule: If there is inheritance between classes then Constructor chaining is mandatory(Rule of Java)

Example: Super

```
class A
{ A()
    {
        System.out.println("Constructor of A class");
    }
}
```

```
class B extends A
{ B()
     {
          super();
          System.out.println("Constructor of B class");
     }
}
class C extends B
{ C()
    { super();
         System.out.println("Constructor of C class");
     }
}
class SuperMain
{
     public static void main(String[] args)
     {
          System.out.println("Main Starts");
          C rv=new C();
          System.out.println("Main Ends");
     }
}
```

Output:

```
Main Starts
Constructor of A class
Constructor of B class
Constructor of C class
Main Ends
```

Note: If there is inheritance between classes and if the Programmer has not developed super classing statement inside the constructor then there will be a default Super calling constructor statement developed by the Compiler
i.e.,Super();Default Super calling statement will always be without “Arguments”

Example: Compiler providing Default Super calling statement

```
package corejava; class D
{ D()
     {
          System.out.println("From D Cosntructor");
     }
}
class E extends D
{ E() {
          //super(); //->Deafult Super calling(Commented Delibrately)
          System.out.println("From E constructor");
     }
```

```
}
class SuperMain1
{
    public static void main(String[] args)
    {
        System.out.println("Main starts");
        E rv=new E();
        System.out.println("Main Ends");
    }
}
```

Output:

```
Main starts
From D Cosntructor
From E constructor Main Ends
```

Super(parameter): This will call the immediate super class constructor with the matching arguments. **Example:**

```
class H
{
    H(int a)
    {
        System.out.println(" from H(int a) constructor");
    }
}
class I extends H
{ I()
    {
        super(90);// will not work when double/flaot vlue is used System.out.println("
        from I(int a) constructor");
    }
}
class Supermain2
{
    public static void main(String[] args)
    {
        I rv=new I();
    }
}
```

Output:

```
 from H(int a) constructor from
 I(int a) constructor
```

Rules for Super class:

1) Super Calling Statement should always be the first statement inside the class.
2) We can develop maximum one super calling statement inside the constructor.
3) Super Calling Statement can be used to call only constructor.

4) If the Programmer has not developed either this() or super(), then compiler will develop the default Super Calling Statement.
5) If the Programmer writes this calling Statement then there will be no super().

Example: class M

```
{
    M()
    {
        System.out.println("From M()");
    }
}
class N extends M
{
    N(int a)
    {
        System.out.println("From N()" +a);
    }
    N()
    { this(10);
        System.out.println("From N()");
    }
}
class O extends N
{
    O()
    {
        System.out.println("From O()");
    }
}
class SuperMain3
{
    public static void main(String[] args)
    {
        O rv=new O();
    }
}
```

Output:

```
From M()
From N()10
From N()
From O()
```

Similarity between this and Super calling statement.

1. Both are used to call Constructor.
2. Both should be first statement inside the constructor.
3. Both can be developed only inside constructor.

4. Both this and super Calling Statement cannot be developed inside a constructor simultaneously at a time.
5. Both this and Super Calling Statement cannot be developed multiple times.

Differences

this callin statement	**super Calling Statement**
1. Is used to call Current class constructor.	1. Is used to call Super class Constructor.
2. will be used in case of Constructor Overloading.	2. Will be used in case of inheritance.
3. will be no "default" this calling statement by compiler	3. Compiler develops a default Super Calling Statement(if there is inheritance).
4.Through this calling statement we can utilize initialization code of another constructor.	4.Through Super Calling Statement we can achieve Constructor chaining.
5. this calling statement is not mandatory.	5. Super Calling Statement is mandatory in case of inheritance.

Example: class Sample//Sample is
Super class

```
{
    int empid;
    Sample(int empid)//Constructor
    {
        this.empid=empid;
    }
}
class Demo extends Sample// Demo is Sub class
{
    Demo(int empid)//Constructor
    {
        super(empid);
    }
}
class SuperMain4
{
    public static void main(String[] args)
    {
        Demo rv1=new Demo(1);
        System.out.println("The value of Empid is: " +rv1.empid);
        Demo rv2=new Demo(2);
        System.out.println("The value of Empid is: " +rv2.empid);
    }
}
```

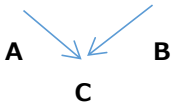
Output:

```
The value of Empid is: 1
The value of Empid is: 2
```

----------**Object class**

1. Object class is the built-in class in Java.
2. Object class is the Super most class for any class in java automatically.
3. Any class is the sub class to Object class.
4. Object class is the first class in any program.
5. All subclass inherits from object class.
6. Object class contains only one Default constructor with No arguments.
7. There are lot of non-static methods already developed in Object class which should be part of every object in java.
8. Any object of java will have all the non-static methods of object class.
9. The mandatory features which should be available in every object of java is developed in object class. Ex: finalaize(),notify(), etc.,

Why Multiple Inheritance is not possible in Java through class.



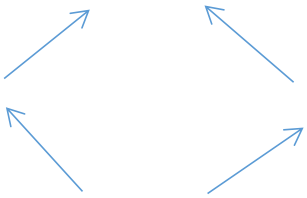
Multiple Inheritance is not possible bcz, in case of MI as 1 class(sub class) will be inheriting from Multiple immediate Super classes then to maintain chaining we should develop 2 or more super calling statement in sub class which is against the rule of java.

Multiple Inheritance

1. Multiple Inheritance means One class inheriting from multiple “immediate” Super class.
2. If there is inheritance between classes, constructor chaining is mandatory.
3. Constructor chaining can be achieved through Super calling statement.

What is Diamond Problem.

If Multiple inheritance is supported in java then it could create a Diamond problem or Path Issue i.e., in case of Multiple inheritance One class will have multiple paths to object class, so same feature of objectc lass might be inherited to same class multiple times in multiple paths, which will create a ambiguity of method usage. This is called as Diamond problem.



8. CONSTRUCTOR

Definition:

Constructor is a special method which is used to construct a object or create a objectmfor a class and cannot return or does not have Return type.

Whenever a object or class is created using new operator constructor will be executed. Contructor does not contain arguments.

Why Contructor?

We can use constructor to create object, without constructor we cannot create Object.

Rules to develop constructor:

1) Constructor named should be exactly same as Class name.
2) Constructor should not have return type, should always be non-static should not return any value.
3) Constructor is also called as Non-static initializer, which is iused ti initialize the object of a class.
4) No static and return type allowed.
5) Constructor should b e there for every class.
6) If the programmer has not developed a constructor for class then there will be default constructor in that class.

Default Constructor:

The constructor developed by the compiler at compile-time is called as Default constructor. If a programmer has nit developed a for class then compiler will develop the default constructor.

Differences:

Methods	Constructor
1) Method can have any name, it can have a class name also.	1) Constructor name Should be similar to the Class name.
2) Method must have return type atleast Void	2) Constructor should have Return type not even void.
3) Method can be static or non-static.	3) Constructor should always be non-static
4) Method may or may not return value.	4) Constructor cannot return value.
5) Java does not provide Default Constructor method.	5) Java provides Default Constructor.
6) Method cannot be used for Object creation.	6) Constructor will be used for object creation using new operator.

Note: Steps in Object creation

1) New operator will laoocate the new address space randomly in the heap memory.
2) All the Non-Static members of class will be loaded onto the address space with default values.
3) Constructor body will be executed on Stack memory like a method. 4) Object address will be assigned to Reference variables.

Why Programmer should write Constructor if java provides Default Constructor?

If the programmmer needs to initialize the instance variable then programmer can perform the initialization in Constructor.

If the programmmer wants to perform some start up activities whenever a object is created for the class, then programmer will reuse the Constructor body and all the start up code will be developed inside the Constructor.

Types of Constructor:

1) Parameterized Constructor: The Constructor which is developed with some arguments.

Example: A(int I)

2) No-Arguments Constructor: The Constructor which is developed without arguemnts.

Example:A()

Default Constructor falls under No-Arguments Constructor.

Note:

1) To create the object for a class we should utilize available Constructor of that ckass.
2) If programmer develops a Constructor for class then java does not provide a default Constructor.

Constructor Overloading:

Developing multiple Constructor for a class with the variation in the Data-type, members and position of arguemnts is called Constructor overloading.

Example:

```
public class Simple1
{
    Simple1()
    {
        System.out.println("Running Constructor");
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Simple1 s1=new Simple1();
        System.out.println("Main Ends");
    }
}
```

Output:

Main Starts
Running Constructor Main Ends

Example: public class Simple2

```
{
    Simple2()
    {
        System.out.println("Running Demo1 constructor");
    }
    public static void main(String[] args)
    {
```

```
            System.out.println("Main Starts");
            Simple2 s2=new Simple2();
            System.out.println("------------");
            Simple2 s22=new Simple2();
            System.out.println("Main Ends");
      }
}
```

Output:

```
Main Starts
Running Demo1 constructor
------------
Running Demo1 constructor
Main Ends
```

Example:

```
public class Simple3
{
      /* DefaultConstructor
      Simple3()
      {
      }
      */
      public static void main(String[] args)
      {
            System.out.println("Main Starts");
            Simple3 s3=new Simple3(); //Call to default constructor
            System.out.println("Main Ends");
      }
}
```

Output:

```
Main Starts Main Ends
```

Example:

```
public class Simple4
{
      void Simple4()
      {
            System.out.println("Constructor ?");//MEthod with class name.
      }
      public static void main(String[] args)
      {
            System.out.println("Main Starts");
            Simple4 s4=new Simple4();
            System.out.println("Main Ends");
            System.out.println("-----------");
       }
```

```
}
```

Output:

```
Main Starts
Main Ends
----------
```

Example:

```
public class Simple5
{
    int stdId; Simple5()
    {
        System.out.println("Running Constructor");
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Simple5 s5=new Simple5();
        System.out.println("Value of stdId is :" +s5.stdId);
        System.out.println("--------------------");
        Simple5 s55=new Simple5();
        System.out.println("Value of stdId is :" +s55.stdId);
        System.out.println("Main Ends");
    }
}
```

Output:

```
Main Starts
Running Constructor
Value of stdId is :0
--------------------
Running Constructor
Value of stdId is :0
Main Ends
```

Example: Parametrized Constructor public

```
class Simple6
{
    int empId; Simple6(int id)
    {
        empId=id;
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Simple6 s6=new Simple6(10);
        System.out.println("Employee id is:" +s6.empId);
        Simple6 s66=new Simple6(20);
        System.out.println("Employee id is:" +s66.empId);
```

```
            System.out.println("Main Ends");
        }
}
```

Output:

```
Main Starts
Employee id is:10
Employee id is:20 Main Ends
```

Example:

```
public class Simple7
{
      int stdId;
      String stdName;
      Simple7(int id,String name)
      { stdId=id;
            stdName=name;
      }
      public static void main(String[] args)
      {
            System.out.println("Main Starts");
            Simple7 s7= new Simple7(1,"JSM");
            System.out.println("1st student id is:" +s7.stdId);
            System.out.println("1st student name is:" +s7.stdName);
            System.out.println("------------------------");
            Simple7 s77= new Simple7(1,"SMP");
            System.out.println("1st student id is:" +s77.stdId);
            System.out.println("1st student name is:" +s77.stdName);
            System.out.println("Main Ends");
      }
}
```

Output:

```
Main Starts
1st student id is:1
1st student name is:JSM
------------------------
1st student id is:1
1st student name is:SMP Main
Ends
```

Example:

```
public class Circle
{
      double radius; Circle(double r)
      {
```

```
        radius=r;                       double
        a=3.14*radius*radius;
        System.out.println("Area is:" +a);
    }
    public static void main(String[] args)
    {
        Circle c1=new Circle(2.3);
        Circle c2=new Circle(5.2);
    }
}
```

Output:

Area is:16.610599999999998 Area
is:84.90560000000002

Example:

```
public class Simple8
{
    int stdId; Simple8(int id)
    { stdId=id;
    }
    public static void main(String[] args)
    {
        Simple8 s8=new Simple8(10);
        //If no arguments specified, actaul and formal arguemnts will differ
        System.out.println("Value of stdId is: " +s8.stdId);
    }
}
```

Output: Value of stdId
is: 10

Example:

```
public
class A
{ A()
    {
        System.out.println("From A()");
    }
    A(int a)
    {
        System.out.println("From A(int a)");
    }
    A(double d)
    {
        System.out.println("From A(double d)");
    }
    A(int i, double d)
    {
```

```
        System.out.println("From A(inti,double d)");
    }
    public static void main(String[] args)
    {
        A a1=new A();
        A a11=new A(50);
        A a111=new A(50.1);
        A a1111=new A(50,50.8);
        A a11111=new A(50,60.9);
    }
}
```

Output:

```
From A()
From A(int a)
From A(double d)
From A(inti,double d)
From A(inti,double d)
```

Example:

```
public class Student
{
    String stdName; int
    stdId; int age;
    Student(String name, int id, int a)
    {
        stdName=name;
        stdId=id;
        age=a;
    }
    Student(String name, int id)
    {
        stdName=name;
        stdId=id;
    }
    public static void main(String[] args)
    {
        Student s1=new Student("JSM", 10,90);
        System.out.println("1st student id is:" +s1.stdId);
        System.out.println("1st student Name is:" +s1.stdName);
        System.out.println("1st student age is:" +s1.age);

        Student s2=new Student("SMP", 20);
        System.out.println("1st student id is:" +s2.stdId);
        System.out.println("1st student Name is:" +s2.stdName);
    }
}
```

Output:

1st student id is:10
1st student Name is:JSM
1st student age is:90
1st student id is:20 1st
student Name is:SMP

Can we Override constructor?

NO, because constructotr will not be inherited to sub class but for method overriding inheritence is mandatory.

Can we Override main method?

NO, because main is static method, static method will not be inherited to sub class and for method overriding inheritence is mandatory.

Can we Override Static members?

NO, because static ethods will not be inherited to subc class but for method overriding inheritence is mandatory.

10. METHOD OVERRIDING

Definition:

Changing the implementation of super class method in the sub class according to the needs of the sub class is called Method overriding.

To achieve Method Overriding 3 things are mandatory, 1.
Inheritance.

2. Non-Static Methods.
3. Signature or Methods Declaration should be same.

Whenever we perform Method overriding super class implementation ofthat method will be lost or masked in the sub class I.e., we will get he latest implementation of sub class.

Advantages of Method Overriding:

1. We can Improve Performance.
2. We can achieve, standardization, Run-time Polymorphism, abstraction.

Example:

```
class C
{
    void test1()
    {
        System.out.println("Running test1 of A class");
    }
}
class D extends C
{
    void test1()
    {
        System.out.println("Running test1 of B class");
    }
}
class Main1
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        D rv=new D();
        rv.test1();
        System.out.println("Main Ends");
    }
}
```

Output:

```
Main Starts
Running test1 of B class Main
Ends
```

Note:

1. Always Overridden methods will be a original method of super class, just implementation will be changed in the class.
2. While object creation itself overriding happens. Later objects address will be assigned to the reference(ex:A rv=new A())

Example: class A

```
{
    void test1()
    {
        System.out.println("Super class Implementation");
    }
}
class B extends A
{
    void test1()
    { super.test1();
        System.out.println("Sub class Implementation");
    }
}
public class Main2
{
    public static void main(String[] args)
    {
        A rv=new B();
        rv.test1();
    }
}
```

Output:

Super class Implementation Sub class Implementation

Example: class Person

```
{
    String name; int age;
    Person(String name,int age)
    {
        this.name=name;
        this.age=age;
    }
    void display()
    {
        System.out.println("The name is:" +name);
        System.out.println("The age is:" +age);
    }
}
class Trainer extends Person
```

```
{
    String subject; double
    salary;
    Trainer(String name,int age,String subject,double salary)
    {
        super(name,age); this.subject=subject;
        this.salary=salary;
    }
   void display()
    { super.display();
        System.out.println("The subject is:" +subject);
        System.out.println("The salary is:" +salary);
    }
}
public class Main3
{
    public static void main(String[] args)
    {

        Trainer tr1=new Trainer("Jo",90,"Java",500000); tr1.display();
        Trainer tr2=new Trainer("J",90,"SQL",500000); tr2.display();
    }
}
```

Output:

```
The name is:Jo
The age is:90
The subject is:Java
The salary is:500000.0
The name is:J
The age is:90
The subject is:SQL
The salary is:500000.0
```

this Keyword:

1) this is a keyword.
2) this will be pointing to the current object under execution.
3) this will be holding the current object address.
4) To access current instance members or object members we should use this keyword.
5) This is also called as “Default Reference variable” of java.
6) To differentiate Global variable with the local variable we can use this keyword.
7) Ex: if a local variable and a Global variable is having the same name “Empid”, then to access the global variable we should use this keyword.
8) This keyword can be used only in non-static context.

Business class and Main class:

1) In one java file, we can develop multiple class.
2) The class in which we develop Biz logic, is called Business class.
3) The class in which Main method is developed is called as Main class.
4) Whenever we are having multiple classes in a java file then we should save the file with main class name.
5) Whenever we compile any java file which is associated with multiple classes then as a result of compilation we will get multiple class file respective to the basic class name. 6) We should run the .class file which is having the Main method.

this calling statement: tcs, is used to call one constructor from another constructor of the same class in case of Constructor Overloading.

Generally for 1 object creation, only 1 constructor shall be executed. But if we want to execute multiple constructor of the same class for only one object creation we can use this() calling statement.

this(parameter) will call the constructor of the same class with matching parameters.

Advantage:

If we cant to use the initialization code written is another constructor of the same class , then we make use of tcs.

Rule to develop the tcs:

1. tcs, should always be the first statement inside a constructor.
2. We cannot develop multiple tcs, inside a single constructor.
3. We can write maximum 1 tcs, inside a constructor.
4. tcs, an be developed only inside a constructor.
5. tcs, is used to call only constructor of the same class.
6. If we want to call constructor from another constructor we should use tcs,. we cannot use constructor name because we can use the constructor name only at the time of object creation.

Example: public

```
class Z
{
    Z()//Constructor
    {
        System.out.println("from A()");
    }
    Z(int a)//Constructor
    { this();//tcs
        System.out.println("from A(int a)");
    }
    public static void main(String[] args)
    {
        Z rv=new Z(90);
    }
}
```

Output: from A()
from A(int a)

Example:

```
public class Demo
{
    Demo()
    {
        System.out.println("From demo()");
    }
    Demo(int a)
    { this();
        System.out.println("From demo(int a)");
    }
    Demo(int a, double d)
    {
        this(90); //when no parameter given, above Demo(int a) constructor will not be called.
/*But other two constructor will be called, when this(90) constructor is not mentioned here, then
current constructor will be executed not the above 2*/
        System.out.println("From demo(int a, double d)");
    }
    public static void main(String[] args)
    {
        Demo rv = new Demo(20, 20.5);
    }   }
```

Output:

```
From demo()
From demo(int a) From
demo(int a, double d)
```

Example:

```
public class Sample
{
    Sample(int a)//Constructor
    {
        System.out.println("from sample(int a)" +a);
    }
    Sample(int a, double d)//Constructor Overloaded
    { this(a);
      System.out.println("From Sample(int a, double d)" +a+ d);
    }
    public static void main(String[] args)
    {
        Sample rv=new Sample(90,20.5);
    }
}
```

Output:

from sample(int a)90 From Sample(int a,
double d)90 20.5

Example:

```
public class Student
{
    String stdName; int
    stdId; int age;
    Student(String name, int id)
    {
        stdName=name;
        stdId=id;
    }
    Student(String name, int id, int a)
    {
        this(name,id);
        age=a;
    }
    public static void main(String[] args)
    {
        Student rv1=new Student("J", 10,90);
        System.out.println("1st student id is: " +rv1.stdId);
        System.out.println("1st student Name is: " +rv1.stdName);
        System.out.println("1st student age is: " +rv1.age);
    }
}
```

Output:

```
1st student id is: 10
1st student Name is: J
1st student age is: 90
```

Example:

```
public class Box
{
    int length; int
    breadth; int
    height;
    Box(int l,int b,int h)//parameterized Constructor
    {
        height=h; length=l;
        breadth=b;
    }
    Box(int side)//parameterized & Constructor Overloading
    {
        this(side,side,side); //this is calling above constructor of the same class due to Constructor
Overloading
    }
```

```
    void volume()//Method
    {
        int v=length*breadth*height; System.out.println("Voulme
        is" +v);
    }
    public static void main(String[] args)
    {
        Box b1=new Box(2,3,4); b1.volume();
        Box b2=new Box(2);
/*Volume is 8, bcz this() in Box(int side) is calling the 1st Box and executing
the Voulume();*/ b2.volume();
    }
}
```

Output:

Voulme is 24 Voulme is 8

Example:

```
package MethodOverRiding;
public class Sample1
{
    Sample1()
    {
        System.out.println("From sample1()");
    }
    Sample1(int a)
    {
        System.out.println("from sample(int a)");
        this(); // Call to this, must be the first statement in constructor
     }
    public static void main(String[] args)
    {
        Sample1 S=new Sample1(10);
    }
}
```

Output: call to this must be first statement in constructor **Example:**

```
public class Sample2
{
    int stdId;// Variable
    void test(int stdId)//Method
    {
        System.out.println(stdId);
        System.out.println(this.stdId);//Tcs, inside SOP
    }
    public static void main(String[] args)
    {
        Sample2 s=new Sample2();
```

```
            s.test(10);
      }
}
```

Output

10 0

Example: public class Sample3

```
{
      String name; void
      t(String name)
      {
            System.out.println(name); this.name=name;
            System.out.println(this.name);
      }
      public static void main(String[] args)
      {
            Sample3 s3=new Sample3();
            s3.t("ramesh");
      }
}
```

Output:

ramesh ramesh

Example: class Demo1

```
{
     int c;     int j;
      Demo1(int c,int j)//Constructor
      {
          this.c=c;         this.j=j;
      }
      public static void main(String[] args)
      {
            Demo1 d=new Demo1(10,20);
            System.out.println("d.c=" +d.c); System.out.println("d.j="
            +d.j);

            Demo1 d2=new Demo1(50,60);
            System.out.println("d.c=" +d2.c);
            System.out.println("d.j=" +d2.j);
      }
}
```

Output: d.c=10

d.j=20

d.c=50

d.j=60

Note: We can compile a program without main(), but we cannot execute the program without main().

Note: Method arguments are also called Local Variables.
Note: A method can be called by supplying the value through variables.

Differences:

Static Members	Non-Static Members
1) It has only one copy of instance variables that share among all objects of the class.	1) It has its own copy of instance variables.
2) A static method belongs to the class itself.	2) Non-static methods belongs to each object that is generated from that class.
3) No need to create object, we can directly call classname.methodname()	3) Need to create a object. Classname obj=new Classname(); Obj.methodname();
4) It includes static variables, methods and Static Initialization Block(SIB).	4) It includes non-static variables, methods, constructor and Instance Initialization Block(IIB).

Differences:

Method Overloading	Method Overriding
1) Overloading can be achieved with and without inheritence,i.e., inheritence is not mandatory.	1) Inheritence is manadatory
2) Overloading can be achieved on both Statci and non-static methods.	2) We can override only non-static methods.
3) For overloading, Method name should be same with variations in arguemnts. There is mo restriction on other parts of the method	3) For overridind, complete method declaration should be same including arguments.
4) Through overloading we can achieve compile time polymorphism.	4) Through overriding we can achieve run time ploymorphism.
5) We can overload main method	5) we cannot override main method.
6) we can achieve constructot overloading	6) We cannot overrie constructor

11. ABSTRACT CLASS

Concrete method: The method which will have declaration and definition together is called as Concrete methods.

```
Ex: void test()
{
----------;
```

----------; }

Concrete class: The class in which we can develop concrete methods is called as Concrete class.

```
Ex: class A
{
    void test()
    {
    --------;
    --------;
    }
}
```

Abstract method: The method with just the declaration without definition is called as Abstract method.

Ex: abstract void test()

Abstract class: The class in which we have an option to develop Abstract method is called as an Abstract class.

Note: The Abstract class and Abstract method should be declared with the keyword “abstract”.

```
Ex: abstract class A
    {
        abstract void test();
    }
```

I.Q: We cannot create object of the abstract class or in other words abstract class cannot be instantiated.

Note: Abstract class should always have a sub class and the sub class should give the implementation or definition of the inherited abstract class.

Note: Giving implementation or a new definition to the inherited abstract class in the sub class is also called as Method overriding.

Advantages of Abstract class:

1) Standardization. 2)
Abstraction

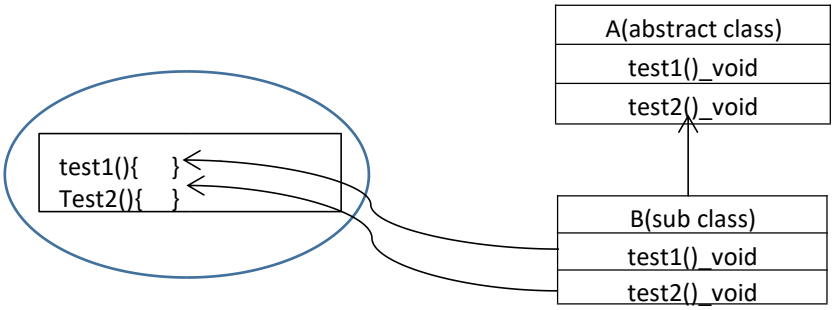
Example:

```
abstract class A
{
    abstract void test1();//only declaration abstract
    void test2();//only declaration
}
class B extends A//inheriting abstract class A
{
    @Override
    void test1()//Declaration
```

```
    {
        System.out.println("test1() is overridden in class B");//Definition
    }
    @Override void
    test2()//Declaration
    {
        System.out.println("test2() is overridden in class B");//Definition
    }
}
public class Tester1
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        //A a=new A();//Class A is abstract and cannot be instantiated B
        b=new B();
        b.test1();
        b.test2();
    }
}
```

Output: Main Starts test1() is overridden in class B test2() is overridden in class B

Demonstration:



Example:

```
abstract class Animal
{
    abstract void move();
    abstract void sound();
}
class dog extends Animal
{
    @Override void
    move()
    {
        System.out.println("dog moves");
```

```
    }
    @Override void
    sound()
    {
        System.out.println("bow bow");
    }
}
class cat extends Animal
{
    @Override
    void move()
    {
        System.out.println("cat moves");
    }
    @Override
    void sound()
    {
        System.out.println("meow meow");
    }
}
public class Tester2
{
    public static void main(String[] args)
    {
        //Animal a=new Animal();//Animal class is abtract and cannot be instantiated
        dog d1=new dog(); d1.move(); d1.sound(); cat c1=new cat(); c1.move();
        c1.sound();
    }
}
```

Output: dog
moves
bow bow
cat moves
meow meow

Example:

```
abstract class college
{
    abstract void gridview();
}
class faculty extends college
{
    String name; int yoe;
    faculty(String name, int yoe)// abstract class can have a constructor
    {
```

```
            this.name = name;
            this.yoe = yoe;
      }
      @Override
      void gridview()
      {
            System.out.println("the name is " +name);
            System.out.println("the total exp is " +yoe);
      }
}
public class Tester3
{
      public static void main(String[] args)
      {
            faculty f=new faculty("JSM", 2); f.gridview();
      }
}
```

Output:

the name is JSM the total
exp is 2

Example: abstract class

```
Sample
{
      abstract void test1(); abstract
      void test2();
}
class Demo extends Sample// sub class Demo should implement all the methods of abstract methods
{
      void test1()
      {
            System.out.println("Test1() overridden in demo class");
      }
}
//void test2() is not inherited and implemented, so CTE. public
class Tester4
{
      public static void main(String[] args)
      {
            Demo d1=new Demo();
          d1.test1(); d1.test2();
      }
}
```

Output:

Abstract_Class.Demo is not abstract and does not override abstract method test2() in Abstract_Class.Sample

What is Abstract class?

The class which is declared with the keyword “abstract” is called as abstract.

Abstract class can contain both Abstract methods and Concrete class or only abstract methods or only concrete methods.

We cannot create object of the abstract class or abstract class cannot be instantiated, it should always have a sub class and the sub class should follow 2 rule,

Rule1: The sub class should override all the inherited abstract methods.

Rule2: If the sub class does not override all the abstract methods then the sub class should be declared as abstract.

Through abstract class we can achieve standardization and abstraction.

Abstract class in not 100% abstract because abstract class can contain concrete methods also.

Ex1: Animal is a concept and it defines the standard characteristics which should be a part of all animal types. All the type of animal will have the standard characteristics with slight variation In behavior.

In this scenario we can go for Abstract class i.e., every animal will have moment and sound characteristics, but implementation differ.

Ex2:While developing the project , all the standard features should be developed with abstract method in abstract class and underlying related class will be implement the standard features based on variation. Here through abstract class we can put a restriction for all the sub class to give the implementation for the standard features.

I.Q: Can we declare an abstract method as static.

No, because static methods cannot be overridden but abstract methods should be overridden. So static method cannot be abstract.

I.Q: Can we declare a abstract method as Final.

No, because final methods cannot be overridden and abstract methods should be overridden.

I.Q: Can we declare a abstract class as Final.

No, because final class cannot have any sub class but abstract class should have a sub class to override its abstract methods.

I.Q: Will abstract class have constructor.

Yes, abstract class will have constructor to maintain constructor chaining.

12. FOUR JAVA TYPES

There are 4 java types, they are:

1) Class
2) Interface
3) Annotation
4) Enumeration

4) Enumeration:

They are used to group the fix number of constants.

Ex: Days in a week, keys of keyboard,months in year. Through enumeration we can achieve uniformity.

Syntax:

```
enum enumname
{

}
```

Example:

```
enum months;
    {

    }
```

5) Annotation:

Definition: Instructions given by programmer to compiler specifying what a compiler should do during compilation.

or

They are used to give implementation to the developer, compiler and run time environment.

Annotations in Java are:

1) @override
2) @Supresswarnings 3) @deprecated

1) @override:

Helps us to check whether we are overriding a method or not.

Example:

```
public
class H
{
    public void test()
    {
        System.out.println("from test()");
    }
}
public class I extends H
{
    @override //complier will help in typo
    public void test1()
```

```
    {
        System.out.println("hello");
    }
    public static void main(String[] args)
    {
        I i1=new I();
        i1.test();
    }
```

Output:

from test()

2) @Supresswarnings

Helps us to suppress warning messages in the program

Example:

```
public
class G
{
    @SuppressWarnings("unused")
    public static void main(String[] args)
    { int i;
    }
}

public class F
{
    //@SuppressWarnings("unused");
    public static void main(String[] args)
    { int i;
    }
}
```

3) Deprecated:

Helps us to notify that a particular method is not in use.

Example:

```
public
class E
{
    public static void main(String[] args)
    {
        E e1=new E();
        e1.test();
    }
    @Deprecated
    public void test()
    {
        System.out.println("From test");
    }
```

```
}
```

Output:

From test

Before running the program annotations will be executed.
Using a built-in annotation override, developer will be notified about proper overloading using annotation suppress warning we can give information to compiler not to display some kind of warning.
Using “TESTNG” annotation we can give info to Run-time information about test cases.

Syntax:

```
@interface annotationname
{

}
```

Example:

```
@interface A
{

}
```

Example: interface A

```
{
    void test1(); void
    test2();
}
class B implements A
{
    @Override
    public void test1()
    {
        System.out.println("test1() overridden in Class B");
    }
    @Override
    public void test2()
    {
        System.out.println("test2() overridden in Class B");
    }
}
public class Tester1
{
    public static void main(String[] args)
    {
        //A rv=new A();//we cannot create objects of interface and A is abstract and cannot be
instantiated B b1=new B();
        b1.test1(); b1.test2();
```

```
    }
}
```

Output:

```
test1() overridden in Class B test2()
overridden in Class B
```

Example: interface TV

```
{
    void display(); void sound();
    void remote(int channel);
}
class sony implements TV
{
    @Override
    public void display()
    {
        System.out.println("sony led display");
    }
    @Override
    public void remote(int channel)
    {
        System.out.println("Sony is in channel number" +channel);
    }
    @Override public void
    sound()
    {
        System.out.println("sony dts sound system");
    }
}
public class Tester2
{
    public static void main(String[] args)
    {
        sony s1=new sony();
        s1.display(); s1.remote(10);
        s1.sound();
    }
}
```

Output:

```
sony led display Sony is in
channel number10 sony dts
sound system
```

Example: interface D

```
{
```

```
    void test1();
}
interface E
{
    void test2();
}
class F implements D, E
{
    public void test1()
    {
        System.out.println("test1() overridden in test1()");
    }
    public void test2()
    {
        System.out.println("test2() overridden in test2()");
    }
}
public class Tester3
{
    public static void main(String[] args)
    {
        F f1=new F(); f1.test1();
        f1.test2();
    }
}
```

Output: test1() overridden in
test1() test2() overridden in
test2()

Example: class G

```
{
    void test1()
    {
        System.out.println("test1() of class F");
    }
}
interface H
{
    void test2();
}
class I extends G implements H
{
    @Override public void
    test2()
    {
        System.out.println("test2() overridden in class H");
```

```
    }
}
public class Tester4
{
    public static void main(String[] args)
    {
        I i1=new I(); i1.test1();
        i1.test2();
    }
}
```

Output: test1() of class F test2()
overridden in class H

Example: interface

```
exam
{
    void percentage();
}
class student
{
    String name; int
    m1,m2,m3;
    student(String name,int m1,int m2,int m3)
    {
        this.name=name;
        this.m1=m1; this.m2=m2;
        this.m3=m3;
    }
    void display()
    {
          System.out.println("the name is " +name);
          System.out.println("the marks1 is: " +m1);
          System.out.println("the marks2 is: " +m2);
        System.out.println("the marks2 is: " +m3);
    }
}
class Result extends student implements exam
{
    Result(String name,int m1,int m2,int m3)
    {
        super(name, m1,m2,m3);//call to student constructor of student class
    }
    public void percentage()//interface
    {
        int total=m1+m2+m3; int
        percent=(total*100)/300;
```

```
            System.out.println("percentage of " +name+ "is: " +percent);
        }
}
public class Tester5
{
        public static void main(String[] args)
        {
            Result r1=new Result("JSM",99,99,99); r1.display();//from
            class
            r1.percentage();//from interface
        }
}
```

Output:

the name is JSM the marks1
is: 99 the marks2 is: 99 the
marks2 is: 99 percentage
ofJSM is: 99

13. INTERFACES

This is a Java type which is 100% abstract. Through interface we can achieve multiple inheritance in java up-to some extent.

Through interface we can achieve standardization and abstraction.
Through interface we can achieve 100% abstraction (because interface is abstract).
We can create Object of Interface.

Interface should always have sub class and the sub class should follow 2 contract or rule.
Rule 1: The sub class should override all the method of interface.
Rule 2: If the sub class does not override all the method of interface, then the sub class should be declared as abstract.

Note:

1) If a class is inheriting from a interface then we should use “implements” keyword.
2) In interface all the methods will be automatically abstract i.e., no need to use abstract keyword.
3) In interface all the methods will be automatically public.
4) In interface we can develop abstract method.
5) Interface will not have constructor because constructor cannot be abstract.
6) Interface will not inherit from object class.
7) If a class is inheriting from interface, no need of constructor chaining(because constructor will not be there)

Explain how multiple inheritance can be achieved through interface in java?

If a class is inheriting from interface no need of constructor chaining so need to write 2 super calling statement and there will be no diamond problem because interfaces will not be inheriting from object class, so through interfaces we can achieve Multiple interface I java up to some extent, up-to some extent because interface cannot have any coding or interface methods cannot have body or implementation.

Note:

1) If a class is inheriting from another class we should use extends keyword(and to represent the inheritance through diagram we should use solid line).

2) If a class is inheriting from interface, we should use implements keyword(we should use dotted line to represent inheritance through diagram).

3) If a interface in inheriting from another interface we should use extends keyword(we should use solid line to represent the interfaces).

I.Q: Can a interface inherit from another interface?

Yes, one interface can inherit from another interface and to represent interface we should use extends keyword.

I.Q: Can one interface inherit from multiple interface.

Yes, it is possible again we should use extends keyword.

I.Q: Can one interface inherit from concrete class or abstract class?

No, because interface is 100% abstract and in this case it might inherit concrete methods.

What is Marker interface?

The interface developed without any methods is called as marker interface.
Through marker interface we can give special information to run time environment or to take special information from run time environment we use marker interface.
Ex:Clonable, serialize-able are built-in marker interface.

If a class is sub class to clone-able marker interface then only we can achieve cloning through clone method.

If a class is sub class to serialize-able maker interface then only we can write the attributes of that class into the file(This is called as Serialization)

Difference:

Abstract class	Interface
1) Abstract class is not 100% abstract.	1) Interface is 100% abstract.
2) In Abstract class we can have constructor.	2) Interface will not have any constructor.
3) Abstract class will inherit from object class.	3) Interface will not inherit from object class.
4) Through Abstract class we cannot achieve multiple inheritance.	4) through interface we can achieve multiple inheritance.
5) If a class is inheriting from abstract class we should use extends keyword.	5) if a class is inheriting from interface then we should use “implements” keyword.
6) in abstract class, Abstract methods should be declared with the keyword “Abstract”.	6) In interface we need not declare abstract methods with keyword abstract(because it will be automatically abstract)
7) in Abstract class, method will have programmer level access.	7) In interface, all the methods will be public.
8) in Abstract class, variables will not be automatically Final.	8) In interface, all the variables will be automatically public static and final.

9) In Abstract class , we can declare a variable and initialize later.	9)In interface, variables should be initialized while declaration itself
10) In Abstract class, we can develop concrete method and that concrete methods can be static so in abstract class we can develop static method also.	10) In interface we cannot develop concrete methods i.e., we can develop only abstract method. Abstract methods cannot be static so in interface we cannot develop static members.
11) In Abstract methods, we can develop concrete methods and those concrete methods can be final. So in abstract class we can develop final methods.	11) In Interface we cannot develop final methods because it is pure abstract and abstract methods cannot be final.
12) If a class is inheriting from abstract class constructor chaining is required.	12) If a class is inheriting from an interface constructor chaining is not required.

Note:

1) Method can return an object as output.
2) While developing program if you want to return an object from a method then that methods return type should be class name.
3) In other words, if a method return type is a class name it should or will return the object address of that particular class name.

Ex: if the methods return type is Sample then it should or will return the object of Sample class.

14. CALL BY VALUE AND CALL BY REFERENCE

There are two types pf method invocation:

1) Call by Value:
2) Call By Reference:

1) **Call by Value:** Calling the method by supplying the primitive data or simple data like 10, 10.5 etc., is called as Cal by value type of Method invocation.

Ex: test(90);//Supplying the value directly. int
a=90;
Test(a);// supplying the value via variable.

2) **Call by Reference:** Calling a method by supplying the object reference is called as call by reference type of method invocation.

Ex: A rv=new A();
Test(rv);

Note:

1) Methods can also receive objects as input and it can perform the operation on the contents of the object.
2) If any methods arguments is reference variable of sub class then we should supply the object address(reference) of that particular class.

15. CASTING(TYPE & OBJECT) Casting means Conversion.

Type Casting

Definition: Type casting means converting the data or objects from one type to another type is called Type casting.

There are 2 types of statements:

1) Homogeneous Statements:

The statements in which there will be no type mismatch are called as Homogeneous statements.

Ex: int a = 10;

Here we are storing integer data "10" in integer type variable 'a'. So the above statement is called as Homogeneous statement.

2) Heterogeneous Statements:

The statements in which there will be type mismatch are called as Heterogeneous statements.

Ex: int a = 10.5;

Here we are trying to store fractional value of double data in integer type variable 'a'. So there is a mismatch in the data and type of variable the above statement is called as Heterogeneous statement.

There are 2 types of Type Casting:

1. Primitive Casting
2. Object Casting

1) Primitive Casting:

Definition: Casting the data from one type to another primitive type is called as Primitive Casting.

Ex: double a=(int) 10.5;

Here we are converting the primitive double data into integer type which is called as Primitive Casting.

Example:

```
public
class A
{
    public static void main(String[] args)
    {
        int a=(int) 10.9;
        System.out.println("value of a is " +a); double
        d=(double) 10;
        System.out.println("value of d is " +d);
    }
}
```

Output:

```
value of a is 10 value of d
is 10.0
```

Note:

1) Out of 8 primitive data type all the 6 number related data types will support for Primitive casting.

2) There will be one ASCII conversion between char and int. 3) Boolean will not support any type of casting.

There are 2 types of Primitive Casting operation:

1) Widening Operation:

Converting the data from lower primitive type to any one of the higher primitive type is called as Widening Operation.

a. Auto-Widening Operation: Compiler performing the widening operation automatically at compile time is called as Auto-widening or Implicit Widening.

b. Explicit-Widening Operation: Programmer performing widening operation is called as Explicit Widening.

2) Narrowing Operation:

Converting the data from higher primitive type to anyone of the lower primitive type is called as Narrowing operation.

a. Explicit-Widening Operation: Programmer performing the narrowing operation explicitly while developing the code is called as Explicit Narrowing.

b. Auto-Narrowing Operation: There is no concept of Auto-Narrowing, i.e., compiler will not perform narrowing operation automatically because might be loss of data.

Example:

```
public
class B
{
    public static void main(String[] args)
    {
        double d=10;
        System.out.println("value of d is " +d); int
        a=(int)10.5;
        System.out.println("Value of a is " +a);
    }
}
```

Output:

```
value of d is 10.0 Value of a
is 10
```

Example:

```
public
class C
{ static int test()
    {
        double d=10.5;
        return(int)d;
    }
    public static void main(String[] args)
    {
        System.out.println(test());
```

```
    }
}
```

Output: 10

Note:

1. In case of Method overloading, appropriate version of the method will be called.
2. If appropriate version is not available then method with Higher version will be called (widening).

Example:

```
public
class D
{
    static void test(int a)
    {
        System.out.println("test(int a)");
    }
    static void test(double d)
    {
        System.out.println("test(double d)");
    }

    public static void main(String[] args)
    {
        test(10);
    }
}
```

Output: test(int a)

Example:

```
public
class E
{
    static void test(int a, double d)
    {
        System.out.println("test(int a, double d)");
    }
    static void test(double d, int a)
    {
        System.out.println("test(double d, int a");
    }
    public static void main(String[] args)
    { test(10,10);
        test(20.5,20);
    }
}
```

Output:

The method test(int, double) is ambiguous for the type E

Example: public
class F
{
public static void main(String[] args)
{
int a=5; int b=2; double
res=(double)a/b;
System.out.println("result is " +res);
}
}

Output: result is
2.5

Example: public
class G
{
static void percentage(int m1,int m2, int m3)
{
int total=m1+m2+m3; double
per=((double)total*100)/300;
System.out.println("percentage=" +per);
}
public static void main(String[] args)
{
percentage(99, 97, 96);
}}
Output: percentage=97.33333333333333
done
Example: public
class H
{
static void area(double radius)
{
double res=3.14*radius*radius; System.out.println("result
is " +res);
}
public static void main(String[] args)
{ area(2.5);
area(2);
}
}
Output:
result is 19.625 result is
12.56

Example: public
class I
{
static double purchaseAmount=95.5; static double
receivePayment(double payedAmount)
{
double change=payedAmount-purchaseAmount;
return change;
}
public static void main(String[] args)
{
double change=receivePayment(100);
System.out.println("Change to be given to the customer is " +change);
}
}

Output:

Change to be given to the customer is 4.5

Example: public
class J
{
public static void main(String[] args)
{
System.out.println(Integer.MAX_VALUE);
System.out.println(100);
System.out.println(99999);
System.out.println(989893898);
}
}

Output:

2147483647
100
99999
989893898

```
public class K
{
    public static void main(String[] args)
    {
        long l=9888983498329L;
        System.out.println("the value of l is " +l); long
        l1=100;
        System.out.println("the value of l1 is " +l1);
    }
}
```

Output:

the value of l is 9888983498329 the
value of l1 is 100

Example: public

```
class L
{
    public static void main(String[] args)
    {
        float f=10F;
        System.out.println("value of f is " +f);
    }
}
```

Output:

value of f is 10.0

Example: public

```
class M
{
    public static void main(String[] args)
    {
        double res=10.0/3;
        System.out.println("The result is " +res); float
        f=(float)res;// Narrowing double value of res to float
        System.out.println("value of f is " +f);
    }
}
```

Output:

The result is 3.3333333333333335 value of
f is 3.3333333

Example: public

```
class N
{
    static void test(byte b)
    {
        System.out.println("test(byte b)");
```

```
    }
    static void test(int a)
    {
        System.out.println("test(int a)");
    }
    public static void main(String[] args)
    { test(1);
    }
}
```

Output: test(int a)

Example: public

```
class O
{
    public static void main(String[] args)
    {
        char ch='A';
        System.out.println("char " +ch); char
        ch2=90;
        System.out.println("char " +ch2); int
        a='c';
        System.out.println("int " +a); int
        b=4;
        System.out.println("int " +b);
        System.out.println('A'+'B');
    }
}
```

Output: char
Z int 99 int 4
131

Example: public

```
class P
{
    public static void main(String[] args)
    {
        char ch=253;
        System.out.println("a" +ch+ "b" +ch);
    }
}
```

Output:
Aýbý

2) Object Casting

Definition: Casting the object from one class to another class type or interface type is called as Object Casting.

Through Object casting we can achieve run time Polymorphism and we can develop generic functions which can receive any input.

To achieve Object-casting there should be “is-a” relationship or there should be inheritance.

There are 2 types of Object-casting Operation:

1) Up-Casting
2) Down-Casting

1) Up-casting: Converting the sub-class object to super-class type or interface type is called as Up-casting.

Whenever we perform up-casting sub-class features will be hidden we can access only super-class features or parent features.

To achieve Up-casting inheritance is mandatory.

```
class Sample1
{
	void test1()
	{
		System.out.println("test1 of super-class sample1");
	}
}
class Sample2 extends Sample1
{
	void test2()
	{
		System.out.println("test2 of sub-class sample2");
	}
}
class Tester1
{
	public static void main(String[] args)
	{
		Sample1 s1=(Sample1)new Sample2();
		s1.test1();
	//s1.test2();
	}
}
```

Output:

```
test1 of super-class sample1
```

a. Auto Up-casting Operation:

Compiler performing the Up-casting operation automatically at compile time is called as “Auto Up-casting”

a1) Up-casting in case of Multilevel Inheritance:

In case of multilevel inheritance performing up-casting we can access super class features.

Example: class

```
Sample3
{
    int a=10;
}
class Sample4 extends Sample3
{
    double d=10.5;
}
class Sample5 extends Sample4
{
    char ch='z';
}
public class Tester2
{
    public static void main(String[] args)
    {
        Sample4 s4=new Sample5();
        System.out.println("The value of a is " +s4.a);
        //System.out.println("The value of d is " +s4.d);
        //System.out.println("The value of ch is " +s4.ch);

        //Sample3 s3=new Sample5();
        //System.out.println("The value of a is " +s3.a);
        //System.out.println("The value of d is " +s3.d);
        //System.out.println("The value of ch is " +s3.ch);

        Sample5 s5=new Sample5();
        System.out.println("The value of d is " +s4.d);
        System.out.println("The value of Ch is " +s5.ch);
    }
}
```

Output:

```
The value of a is 10
The value of d is 10.5 The
value of Ch is z
```

a2) Up-casting in case of Method Overriding:

You can access overridden methods of super class but we get the latest implementation.

Example: class

```
Sample6
{
    void test()
```

```
    {
        System.out.println("test() super class implementation");
    }
}
class Sample7 extends Sample6
{
    void test()
    {
        System.out.println("test() of sub class implementation");
    }
}

public class Tester3
{
    public static void main(String[] args)
    {
        Sample6 s6=new Sample7(); s6.test();
    }
}
```

Output: test() of sub class implementation

2) Down-Casting: Converting super class object o sub class type is called as Down Casting

Whenever we perform down-casting, sub class features will be hidden.

So to access these hidden features we perform down-casting operation.

To perform Down-casting there should be “is-a” relationship or there should be inheritance to the classes.

Direct down-casting is not possible.(It means performing down-casting without up-casting)

There is no concept if Auto Down-casting, Programming should do it. So it is called as Explicit Casting.

```
class Sample10
{
    void test1()
    {
        System.out.println("Running test1() of class Sample10");
    }
}
class Sample11 extends Sample10
{
    void test2()
    {
        System.out.println("Running test2() of class Sample11");
    }
}
public class Tester4
{
    public static void main(String[] args)
```

```
    {
        Sample10   s10=new   Sample11();
        Sample11       s11=(Sample11)s10;
        s11.test1(); s11.test2();
    }
}
```

Output:

Running test1() of class Sample10
Running test2() of class Sample11

Example:

```
class Sample12
{
    void test1()
    {
        System.out.println("test1()");
    }
}
class Sample13 extends Sample12
{
    void test2()
    {
        System.out.println("test2()");
    }
}
public class Tester5
{
    public static void main(String[] args)
    {
        Sample12   s12=new   Sample12();
        Sample13       s13=(Sample13)s12;
        s13.test1(); s13.test2();
    }
}
```

Output: java.lang.ClassCastException: objectcasting.Sample12 cannot be cast to objectcasting.Sample13 **Example:** class Pendrive implements USBPORT

```
{
    public void read()
    {
        System.out.println("Reading from Pendrive");
    }
    public void write()
    {
        System.out.println("Writing to pendrive");
    }
}
```

```
class Mobiledevice implements USBPORT
{
	public void read()
	{
		System.out.println("Reading from Mobile-device");
	}
	public void write()
	{
		System.out.println("Writing to Mobile-device");
	}
}
class Myclass
{
	static void demo(USBPORT driver)
	{ driver.read();
		driver.write();
	}
}
class Tester6
{
	public static void main(String[] args)
	{
		Myclass.demo(new Pendrive());
		Myclass.demo(new Mobiledevice());
	}
}
```

Output:

Reading from Pendrive
Writing to pendrive
Reading from Mobile-device
Writing to Mobile-device

16. POLYMORPHISM

Definition: Polymorphism is the ability of an object to take on many forms.
The most common use of Polymorphism in OOP, occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic.
In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable.

A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any sub-type of its declared type.

A reference variable can be declared as a class or interface type.

Note: Method binding means Resolution for method calling statement
.
Based on Method binding , Polymorphism can be categorized into two types:
1) Compile-Time Polymorphism. 2)
Run-Time Polymorphism.

1) Compile-Time Polymorphism:

Through method overloading we can achieve Compile-Time Polymorphism.
Here method binding happens at compile time, so it is called as Compile-Time Polymorphism.
It is also called as Static Polymorphism or Early binding.

2) Run-Time Polymorphism:

To achieve Run-Time Polymorphism 3 things are mandatory,

a. Inheritance.
b. Method Overriding
c. Auto Up-casting.

Here method binding cannot be performed by compiler because overriding happens at run time, so it is called as Run-Time Polymorphism.

It is also called as Dynamic Polymorphism or Late binding.

Note: Static Polymorphism is faster than Dynamic Polymorphism.

Example:

```
public class Employee
{
    private String name; private
    String address; private int
    number;
    public Employee(String name,String address,int number)
```

```
    {
        System.out.println("Constructing an Employee");
        this.name = name; this.address =
        address;
        this.number = number;
    }
    public void mailCheck()
    {
        System.out.println("Mailing a check to "+this.name +" "+this.address);
    }
    public String toString()
    {
        return name +" "+ address +" "+ number;
    }
    public String getName()
    {
        return name;
    }
    public String getAddress()
    {
        return address;
    }
    public void setAddress(String newAddress)
    {
        address = newAddress;
    }
    public int getNumber()
    {
        return number;
    }
}

public class Salary extends Employee
{
    private double salary;//Annual salary
  public Salary(String name,String address,int number, doublesalary)
    {
        super(name, address, number); setSalary(salary);
    }
    public void mailCheck()
    {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to "+ getName() +" with salary "+ salary);
    }
    public double getSalary()
    {
```

```
            return salary;
      }
      public void setSalary(double newSalary)
      {
            if(newSalary >=0.0)
            {
                  salary = newSalary;
            }
      }
      public double computePay()
      {
            System.out.println("Computing salary pay for "+ getName());
            return salary/52;
      }
}
public class VirtualDemo
{
      public static void main(String[] args)
      {
             Salary s =new Salary("Mohd Mohtashim","Ambehta, UP", 3,3600.00);
            Employee e =new Salary("John Adams","Boston, MA", 2,2400.00);
            System.out.println("Call mailCheck using Salary reference --");
            s.mailCheck();
            System.out.println("¥n Call mailCheck usingEmployee reference--");
            e.mailCheck();
      }
}
```

Output:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

 Call mailCheck usingEmployee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

17. ABSTRACTION

Definition: Hiding the complexity of the system and exposing required functionality is called as "Abstraction".

Here class implementation will be hidden from user or usage.

Through Interface we can achieve 100% abstraction.

Through Abstract class we achieve up-to 100% abstraction(bcz, it is not 100% abstract).

Advantages of Abstraction: Internal enhancement will be not impact the usage.

To achieve abstraction:

1) Group all the common features in a interface or abstract class.
2) Override all the common features based in some variation on the sub classes.
3) Develop a helper method in a helper class which returns the appropriate object based on the request.

Example1:

When we drive a car, we will not know the internal complex implementation of car, we just will be exposed to the functionality.

Example2:

While sending mail, one composes and sends a mail, but internally there will be lot of complexity which will be hidden.

Example3:

Selenium is complete abstraction with lot of built-in functionality. To get the title from web-page there are lot of complexities, but being the user of selenium to get the title we use getTitle() which is Abstraction.

Example: public interface SWITCH

```
{
    void on();
    void off();
}
public class Bulb implements SWITCH
{
    public void on()
    {
        System.out.println("bulb is turned on");
    }
    public void off()
    {
        System.out.println("bulb is turned off");
    }
}
public class Fan implements SWITCH
```

```
{
	public void on()
	{
		System.out.println("Fan is turned on");
	}
	public void off()
	{
		System.out.println("Fan is turned off");
	}
}
public class MyClass
{
	static SWITCH mymethod(char ch)
	{ if(ch=='f')
		{
			return new Fan();
		}
		else
		{
			return new Bulb();
		}
	}
}

public class Tester1
{
	public static void main(String[] args)
	{
		SWITCH s1=MyClass.mymethod('f');
		s1.on(); s1.off();
	}
}
```

Output:

Fan is turned on
Fan is turned off

18. ENCAPSULATION

Restricting the direct access to the data members and giving indirect access to the data members through the getter and setter method is called as Encapsulation.

While performing encapsulation, the data members will be declared as Private and getters and setters method be declared as Public.

Getter are public method which will give the data and Setter are public method which will manipulate the data.

Conceptually they are getters and setters, but method name need not to be always get and set.Its a standard industry convention to name gets and sets with get and set followed by private data member name.

i.e., get<private data members> set<private data members>

Ex: getBalance()
setBalance()

Getters are also called as Accessors.
Setters are also called as Mutators.
Through encapsulation, we can achieve data security , i.e., we can have complete control over the data which will be manipulating.
Here the Data and code which will be manipulating the data will be binded in the same class.

Java Bean Class

Developing a class with private data members and giving access to those private data members using public service methods getters and setters is called Java Bean Class.

Steps to generate Getters and Setters in IDE.

1. Select private members of a class and R.C.
2. Click on SRC
3. Click on generate Getter and Setter.
4. Select the getters and setters method an dclick on finish.

Example: Balance of a Account holder should always be developed as private data members in programs so that code can access it or manipulate it directly(it should happen through getters and setters method based on Business condition).

Difference b/w Abstraction and Encapsulation?

In Abstraction complexity of the system will be hidden and in encapsulation there will be restriction on direct access.
Ex: Booking the ticket is abstraction.
Unable to Book the already booked ticket is Encapsulation.

Example:

```
package Encapsulation; class Z
{
    private double empSal;
```

```
    Z(double empSal)
    {
        this.empSal=empSal;
    }
    public double getempSal()//Getters
    {
        return empSal;
    }
    public void setempSal(double empSal)//Setters
    {
        this.empSal=empSal;
    }
}
class B
{
    public static void main(String[] args)
    {
        Z z = new Z(2000.5);
        System.out.println("Emp sal is" +z.getempSal());
        //a.empSal=9000.5;//No direct access
        z.setempSal(9000.5);
        System.out.println("Emp sal is" +z.getempSal());
    }
}
```

Output:

```
Emp sal is2000.5
Emp sal is9000.5
```

Example:

```
package Encapsulation;
public class Sample
{
    private int a; private double
    d; Sample(int a, double d)
    { this.a=a;
        this.d=d;
    }
    public int getA()
    {
        return a;
    }
    public void setA(int a)
    { this.a=a;
    }
```

```
    public double getD()
    {
        return d;
    }
    public void setD(double d)
    { this.d=d;
    }
}
class Tester2
{
    public static void main(String[] args)
    {
        Sample s=new Sample(10,10.5);
        System.out.println("value of a " +s.getA()); System.out.println("Value of d " +s.getD());
         s.setA(90);
         s.setD(90.5);
         System.out.println("Value of a is " +s.getA());
         System.out.println("Value of d is " +s.getA());
    }
}
```

Output: value of a
10 Value of d 10.5
Value of a is 90 Value of d
is 90

Example: package
Encapsulation;

```
class Account
{
    private double balance=5000;
    public double getBalance()
    {
        return balance;
    }
    public void setBalance(double tamount)
    {
        if (tamount>0)
        {
            balance+=tamount;
            System.out.println("Transferred" +tamount);
        }
        else
        {
            System.out.println("Invalid amount");
        }
    }
```

```
}
class Tester3
{
    public static void main(String[] args)
    {
        Account a=new Account();
        System.out.println("Balance=" +a.getBalance());
        a.setBalance(1000);
        System.out.println("Balance=" +a.getBalance() +"Rs"); a.setBalance(-
        6000);
        System.out.println("Balance" +a.getBalance() +"Rs");
    }
}
```

Output:

```
Balance=5000.0
Transferred1000.0
Balance=6000.0Rs
Invalid amount
Balance6000.0Rs
```

19. PACKAGES & ACCESS LEVELS

Packages are like folders which are used to group similar kind of programs.

Through packages we can achieve,

1. Encapsulation
2. Logical Division
3. Avoid naming collisions.,i.e., we can have same class name across interfaces/class name.

Note:

1.Package statement should be the first statement in java file.
2.All the source file will be stored in the classes or bin folder.
3.All the sub folder of src are considered packages.
4.Always compilation should be from src folder.
5.Always execution should be from class folder.

Example: package

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

Output: Hello JSM

Example:

```
package Package2; import
Package.A;
public class B
{
    public static void main(String[] args)
    {
        System.out.println("hello Smp");
    }
}
```

Output:

hello Smp

Access Levels

Access levels are used to put restrictions on the members and through access levels we can achieve encapsulation. There are 4 Access levels,

1. Public
2. Protected
3. Default
4. Private

1. Public: Public members will have application level access i.e., public members can be accessed from any program of any packages.

Note: Public is the most visible access levels.

2. Default: Default members will have package level access i.e., default members can be accessed only within the current package programs in which it is created.

Note: If any members is not declared with any others access level then that members will have default access levels. i.e., there is no keyword to members default.

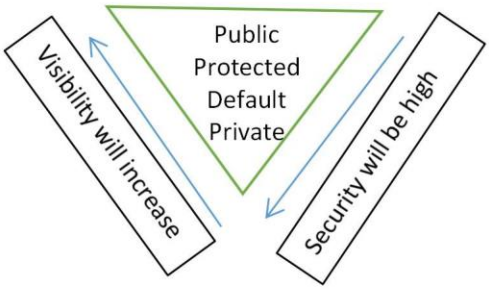
3. Private: Private members will have class level access i.e., private members can be accessed only within the class in which it is created.

Note:In other words, private members cannot be accessed outside the class.

Note:Private members will not be involved in inheritance.

4. Protected: Protected members can be accessed within the package like default access level and it can be accessed outside the package through inheritance.

ess Levels:



To access the members of a class present in other package we should first import the class from other packages by using import statement.

Hierarchy of Access Levels:

Note: 1.

Ex: import pack1.A

2. Import statement should always be developed after package statement(because Package statement should be first statement in the file)
3. We can develop multiple import statemnet in one java file in other words we can use members of different classes available in different package in one file.

Ex: import pack1.A;
Import pack2.sample;

4. If there are lot of classes in one package and if you want to import all the classes of that package then we should use "*".. Ex: import pack1.*;

Note: Protected members can be accessed outside the package only through inheritance and we should create sub class object.

Note:

1. A class cannot be declared with private and protected access levels.
2. A class can have only two access levels Public and Default.
3. Private and Protected access levels are only for members of a class not for the class itself. 4. Public classes can be imported through import statement but Default classes cannot be imported.

5. In java file we can develop maximum one public class nand that public class name should be java file name.(Rule of Java)

I.Q: Can we over ride private non-static methods?

NO ,because private methods will not be inherited to sub class and for method overriding inheritance is mandatory.

I.Q: While overriding a super class method in the sub class, can we change the access level of the method from default to public.

YES, while overriding we can expand the access level from default to protected or public . BUT we cannot weaken the access level from public to protected or default

Example: package

```
Package3;
public class C
{
    public void test1()
    {
        System.out.println("public test1()");
    }
    protected void test2()
    {
        System.out.println("protected test2()");
    }
    void test3()
    {
       System.out.println("    Test3()");
    }
    private void test4()
    {
        System.out.println("Private test4()");
    }
}
package Package4; import
Package3.C;
public class D extends C
{
    public static void main(String[] args)
    {
        D d=new D();
        d.test1();
        d.test2();
    }
}
```

Output: public test1()
protected test2(

20. DESIGN PATTERN

Identifying the best solution for the frequently occurring problem and usung the same solution henceforth(thereafter) if the same problem is reoccurring is called as Design Pattern.

While Development and Automation there will be some frequently occurring problem for those frequently occurring problem some solution are designed by other developer in the industry such type pf solution are called as Design Pattern.

There are lot of Design Pattern some of them are.

1. Singleton class Design Pattern.
2. Doubleton Class Design Pattern.
3. Immutable Class Design Pattern.

1. Singleton class Design Pattern: The class for which only one object can be created such type of class is called as Singleton class.
To Achieve Singleton class Design Pattern we should follow 2 things, a.
Constructor of that class should be Private.
b. There shoul be public, static method which should return only 1 object of that class.

Note:

a) Singleton means Single object of that class referred by multiple references.
b) Through Singleton class design pattern memory can be managed effectively.

2. Doubleton class Design Pattern:The class for which only 2 object can be created such type of classes are called as Doubleton class Design Pattern.

3. Immutable Object Design Pattern: The class object for which state can be changed such type of object are called as Immutable Object Design Pattern. To achieve Immutable object Design pattern, a) Data members should be private. [gets() should be developed without sets()]
b) Variables should be final and that final variable should be initialized through Constructor.

Note:

1. Final Global Variables can be initialized through constructor also.
2. Design PAttern are not any built-in functionality, they are just proposed solution to existing problem.

Example: Singleton Object Design Pattern class

```
A
{
    static A a1=null;
    private A()//1. Private Constructor
    {
        System.out.println("Running Private Constructor");
    }
    void m1()
    {
        System.out.println("non-static method");//Referred multiple times by the Single object
    }
    public static A demo()
    { if(a1==null)
        {
            a1=new A();
            return a1;//2. returns 1 object of a class
        }
        else
```

```
        {
                return a1;
        }
    }
}
public class Tester
{
    public static void main(String[] args)
    {
        A rv1=A.demo(); rv1.m1();
        A rv2=A.demo(); rv2.m1();
        A rv3=A.demo(); rv3.m1();
    }
}
```

Output: Running Private
Constructor non-static method
non-static method non-static
method

Example: Immutable Object Design Pattern class

```
Demo
{
    private int i; private double j;
    Demo(int i,double j)//Constructor
    { this.i=i;
        this.j=j;
    }
    public int getI()
    {
        return i;
    }
    public double getJ()
    {
        return j;
    }
}
public class Tester1
{
    public static void main(String[] args)
    {
        Demo rv1=new Demo(8,8.5);
        System.out.println("rv1.getI()");
        System.out.println(rv1.getI());
        Demo rv2=new Demo(9,9.5);
        System.out.println("rv2.getJ()");
        System.out.println(rv2.getJ());
```

```
    }
}
```

Output:

```
rv1.getI()
8
rv2.getJ()
9.5
```

Example: Immutable Object Design Pattern

```
class Simple
{
    final int a;//State cannot be changed
    Simple(int a)
    { this.a=a;
    }
}
public class Tester2
{
    public static void main(String[] args) {
        {
                Simple rv1=new Simple(10);
                 System.out.println(rv1.a);
                Simple rv2=new Simple(20);
                 System.out.println(rv2.a);
        }
    }
}
```

Output:

```
10
20
```

21. IIB(INSTANCE INITIALIZATION BLOCK)

Definition:

IIB are executed when objects are created, the number of times we create objects same number of times IIB will be called.

IIB are used to initialize all the instance variable in one place and give us better readability of code.

IIB purpose is to initialize all non-static members in one place for readability.

Example:

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

```
    {
        Z z1=new Z();// When object is created
    }
}
```

Output: from IIB

Example: public

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

    }
}
```

Output:

No output because no object is created

Note:

IIB will be called first because first initialize variable then load into object and then Constructor will be called.

Example: public

```
class X
{ X()
    {
        System.out.println("from Constructor");
    }
    {
        System.out.println("from IIB");
    }
    public static void main(String[] args)
    {
        X x1=new X(); // IIB
        System.out.println("from main");
    }
}
```

Output: from IIB from
Constructor from main

Note:

When object is created, initialization begins.
Instance variables can be directly accessed by IIB.

Example:

```
public class V
{ int i;
      {
            i=20;
            System.out.println(i);//instance variable can be directly accessed by IIB
      }
      public static void main(String[] args)
      {
            V v1=new V();
      }
}
```

Output: 20

Note:

We can initialize both static and non-static variable both inside IIB.

Example:

```
public
class U
{ static int i;
      {
            i=40;
            System.out.println(i);
      }
      public static void main(String[] args)
      {
            U u1=new U();
      }
}
```

Output:

40

Example:

```
public
class W
{
      W()
      {
            System.out.println("From constructor");
      }
      {
            System.out.println("from IIB-2");
      }
      {
            System.out.println("from IIb-1");
      }
          public static void main(String[] args)
      {
```

```
            W w1=new W();
        }
}
```

Output: from IIB-2
from IIb-1 From
constructor

22. SIB(STATIC INITIALIZATION BLOCK)

Static block runs first then main() in the sequence, and it does not require any invoking statement and runs only once.

Example: public
class T
{ static
{
System.out.println("from SIB");
}
public static void main(String[] args)
{

}
}

Output: from SIB

Example: public
class S
{ static
{
System.out.println("from SIB");
}
public static void main(String[] args)
{
System.out.println("from main");
}
}

Output: from SIB
from main

Example: public
class R
{ static
{
System.out.println("from SIB-1");
} static
{
System.out.println("from SIB-2");
}
public static void main(String[] args)
{
System.out.println("from main");
}

```
}
```

Output: from SIB-
1 from SIB-2 from
main

Note:

1.IIB can initialize both static & non-static variables, but a SIB can use only static variable initialization.
2.Reference variable can access static and non-static variables.
3.Class can access static anything that is static it belongs to class.
4.In SIB, instance variable initialization is not possible.
5.We cannot initialize non-static variable inside SIB.

Example:

```
public
class Q
{
    {
        System.out.println("IIB");
    } static
    {
        System.out.println("SIB");
    }
    public static void main(String[] args)
    {
        Q q1=new Q();
    }
}
```

Output:

SIB IIB

Example:

```
public
class P
{
    {
        System.out.println("IIB");
    }
    P()
    {
        System.out.println("P()");
    } static
    {
        System.out.println("SIB");
    }
    public static void main(String[] args)
    {
        new P();
```

```
            System.out.println("main");
      }
}
```

Output:

```
SIB
IIB P()
main
```

Example: public

```
class O
{
      {
            System.out.println("1");
      }
      {
            System.out.println("2");
      } static
      {
            System.out.println("3");
      }
      O()
      {
            System.out.println("4");
      }
      public static void main(String[] args)
      {
            O o1=new O();
            System.out.println("main");
      }
}
```

Output:

```
3
1
2 4 main
```

Example: public

```
class N
{
      {
            System.out.println("IIB");
      } static
      {
            new N();// IIB will be called by using the statement
            System.out.println("SIB");
      }
      N()
      {
```

```
        System.out.println("N()");
    }
    public static void main(String[] args)
    {
        //new N();
        System.out.println("main");
    }
}
```

Output:

```
IIB
N() SIB main Ex: public class M
{
static
{
        System.out.println("SIB");
    } static
    {
        System.out.println("Hello");
        new M();// IIB will be called
    }
    {
        System.out.println("IIB");
    }
    public static void main(String[] args)
    {
        new M();
        System.out.println("main");
    }
}
```

Output:

```
SIB
Hello
IIB
IIB
main
```

Example: public

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

```
    }
    {
        System.out.println("IIB");
    }
    L()
    {
        System.out.println("L()");
    }
}
```

//Execution 1. Object is created.

2. then IIB is called.

3. When object is created, constructor is called so print L

Output:

IIB L()

test()

Note:

Inside IIB, can we create a object?

Yes, the program will run and stop abruptly without getting error.

Example:

```
public class K
{
    {
        System.out.println("IIB"); new
        K();
    }
    public static void main(String[] args)
    {
        K k1=new K();
        System.out.println("from main");
    } }
```

Output:

IIB
IIB
IIB
IIB
IIB

Exception in thread "main" java.lang.StackOverflowError

Example:

```
public
class J
{
    {
        System.out.println("IIB");
    } static
    {
```

```
        new J(); //run's first
        System.out.println("SIB");//run's second new
        J();// run's third
    }
    public static void main(String[] args)
    {
        System.out.println("from main");
    }
}
```

Output:

IIB
SIB IIB
from main

23. OBJECT CLASS

1. Object class is the built-in class of java.lang package.
2. Object class is the super most class for any class of java.
3. To use object class in any java file, there is no need to import explicitly through import statement.
4. In every java file, all the class of java.lang package will be imported automatically by the compiler by the following import statement

Import java.lang.*

Object class had 9 non-static method which is part of every object of java. They are:

1. toString()
2. equals()
3. hashcode()

public non-final methods

4. Notify()
5. notifyAll()
6. wait()

public final methods(interthread communication)

7. getclass()Reflections
8. Clone()protected()
9. Finalize()protected()

1. toString()

1. tostring() is a non-static method of object class which will be inherited to every class of java.
2. tostring() will be available to every object of java.
3. **tostring() will give the object information in the string form i.e., it will give the string representation of the object on which it is called.**
4. String representation will contain fully qualified class name with converted hexadecimal numbers for the memory address of object with the separator @.
5. Fully qualified class name means packagename.classname .
6. Whenever we print any reference variable in java, there will be implicit call to tostring() of that object(implicit call means automatically the method will be called). 7. toString() is a public non-final methos of a object class,

8. Any sub class can over ride toString().
9. toString() will always be over ridden to give the detail information of object including attribute and its value.
10. toString() return type is String and its signature is

```
Public string toString()
{

}
```

Note:

1. Java does not supprt pointers so its very secure.
2. We cannot get the exact memory address of object in java, either we get hexadecimal number or hash number but not exact memory address.
3. We cannot increment the reference variable i.e.,rv1++(not possible).

Example:

```
class A
{
    public static void main(String[] args)
    {
        A rv1=new A();
        A rv2=new A();
        System.out.println(rv1.toString());
        System.out.println(rv2.toString());
    }
}
```

Output:

```
ObjectClass.A@15db9742 ObjectClass.A@6d06d69c
```

Example:

```
public class Sample
{
}
class Tester
{
    public static void main(String[] args)
    {
        Sample s=new Sample();
        Sample s2=new Sample();
        System.out.println(s.toString());
        System.out.println(s2.toString());
        Sample s3=new Sample();
        Sample s4=s3;
        System.out.println(s3.toString());
        System.out.println(s4.toString());
    }
}
```

Output:

ObjectClass.Sample@15db9742 ObjectClass.Sample@6d06d69c
ObjectClass.Sample@7852e922
ObjectClass.Sample@7852e922

Example: class

```
Demo
{
}
public class Tester2
{
      public static void main(String[] args)
      {
            Demo d=new Demo();
            System.out.println(d);//Implicit call to toString
            Demo d2=new Demo();
            System.out.println(d2);//Implicit call to toString
      }
}
```

Output:

ObjectClass.Demo@15db9742 ObjectClass.Demo@6d06d69c

Example: class B

```
{
    public String toString()
    {
          return "B class Object";
    }
}
public class Tester1
{
      public static void main(String[] args)
      {
            B b=new B();
            System.out.println(b);
            B b1=new B();
            System.out.println(b1);
      }
}
```

Output:

B class Object
B class Object

2.equals()

1. equals() is a non-static method of object class which will be inherited to every class of java.
2. equals() will be available to every object of java.

3. **equals() will compare 2 objects equality based on memory address, if memory address are same it returns true otherwise false.**
4. equals() is a public non-final methods of object class.
5. Any sub class can override equals().
6. equals() will always be overridden to compare 2 objects equality based on the attributes.

Ex: if you want to compare 2 employee class object based on the employeeId attribute then equals should be overridden..

7. Basically 2 identify the duplicate, equals() will be overridden.
8. equals() return type is Boolean and the signature is public boolean equals(object obj)

```
{

}
```

What is the difference between comparison through equals() and comparison operator overloading(==).

There is no concept of operator overloading in java, i.e., we cannot change the implementation of operator but there is a concept of overloading i.e., we change the implementation of methods.

So equals() can be overridden based on the attribute or comparison but comparison operator(==) cannot be overridden.

Example:

```
package ObjectClass; class Z
{

}
public class Tester4
{
    public static void main(String[] args)
    {
        Z z=new Z();
        Z z2=new Z();
        System.out.println(z.equals(z2));
        Z z3=new Z();
        Z z4=z3;
        System.out.println(z3.equals(z4));
        }
}
```

Output:

```
false
True
```

Example:

```
package ObjectClass;
class Employee
{
    int empId; Employee(int
    empId)
    {
        this.empId=empId;
```

```
    }
    public boolean equals(Object obj)
    {
        Employee e=(Employee)obj;
        if(this.empId=e.empId)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
public class Tester5
{
    public static void main(String[] args) {
        {
            Employee e1=new Employee(10);
            Employee e2=new Employee(10);
            System.out.println(e1.equals(e2));
            Employee e3=new Employee(30);
            Employee e4=new Employee(40);
            System.out.println(e3.equals(e4));
        }
    }
}
```

Output: True
False

3. hashcode()

1. hashcode() is a non-static method of object class which will be inherited to every class of java.
2. hashcode() will be available to every object of java.
3. **hashcode() will give a unique integer number based on the memory address of the object**

i.e., it will give the integer representation of the object.

4. The unique integer number will be generated by using some hashing algorithm
5. The unique integer number is also called as hash number.
6. Hash code is a public non-final methods of object class so any sub class can override it.
7. Always hashcode() will be overridden to generate the has number by using the attributes of the object.
8. hashcode() return type is int.
9. hashcode() signature is

public int hashcode()

```
        {

        }
```

Procedure to compare 2 objects equality upon attribute.

1. Override both equals() and hashcode().
2. Call hashcode() on 2 objects and get the hash number.
3. Compare the hash number.
4. If the hash number are different then objects will be unequal or unique or disjoint.
5. If the hash number are same then objects might be equal or unequal.So in this case to identify duplicate we should call equals().
6. If equals return true object will be equal or duplicate otherwise object will be unequal or unique.
7. If you want to store unique elements by avoiding duplication then we should follow the above procedure.

Example: package

```
ObjectClass; class N
{

}
public class Tester6
{
    public static void main(String[] args)
    {
        N n=new N();
        N n2=new N();
        System.out.println(n.hashCode());//Diiferent output
        System.out.println(n2.hashCode());
        N n3=new N();
        N n4=n3;
        System.out.println(n3.hashCode());//Same Output
        System.out.println(n4.hashCode());
    }
}
```

Output:

```
366712642
1829164700
2018699554
2018699554
```

Example: package

```
ObjectClass;
class M
{ int i; int j; M(int i,int
    j)
    { this.i=i;
        this.j=j;
    }
```

```
    public int hashcode()
    {
        return i+j;
    }
}
public class Tester7
{
    public static void main(String[] args)
    {
        M m1=new M(70,70);
            System.out.println(m1.hashCode());
        M m2=new M(10,10);
            System.out.println(m2.hashCode());
          }
}
```

Output:

366712642

1829164700

24. ASSERT

Assert helps to check the business condition only if the Business condition is true then assert will continue with program execution or else will not execute the program if business condition is false. Assert was introduced in JDK 1.4.

By default assert will not work. i.e., we cannot use it directly. We need to enable it.

Steps to configure Assert:

1) Click on Drop-down of Run Button & Click on Run configurations and Go-to Arguments tab.
2) In the VM arguments Text-field, enter either "-ea" or "-da" to enable or disable resp.

Example:
```
public
class A
{
    public static void main(String[] args)
    {
        int age=10; assert
        age>20;
        System.out.println("Register your self");
    }
}
```
Output:

Exception in thread "main" java.lang.AssertionError, at asserts.A.main(A.java:8)

Example:
```
public
class A
{
    public static void main(String[] args)
    {
        int age=100; assert
        age>20;
        System.out.println("Register your self");
    }
}
```
Output: Register your
self

Example:
```
public
class B
{
    public static void main(String[] args)
    {
        int age=100; try
        {
            assert age>20;
            System.out.println("Register your self");
        }
        catch(AssertionError e)
```

```
            {
                System.out.println(e);
                System.out.println("You are too young");
            }
        }
}
```

Output:

Register your self

Note:

1) assert creates an exception when condition fails.
2) assert should be handled using try catch blocks.
3) Conditions used in assert should be Boolean value.
4) Multiple assert statement can be used in same class.
5) Statement following assert will execute when assert is true.
6) Statement following assert will not execute when assert is false.

Example:

```
public
class C
{
    public static void main(String[] args)
    {
        assert true;
        System.out.println("hello"); assert
        false;
        System.out.println("Error");
    }
}
```

Output:

```
hello
Exception in thread "main" java.lang.AssertionError,at asserts.C.main(C.java:9)
```

Example:

```
public
class D
{
    public static void main(String[] args)
    { assert test();
        System.out.println("From main()");
    }
    public static boolean test()
    {
        System.out.println("from test()"); return
        false;
    }
}
```

Output:

```
From
main()
```

Example: public
class E
{
public static void main(String[] args)
{ assert test();
System.out.println("from main");
}
public static boolean test()
{
System.out.println("from test"); return
true;
}
}
Output: from test
from main

25. STRING CLASS

1. String is a concrete class in java.
2. String us a built-in class of java.
3. String class is developed to handle string literals or string values.
4. String class is available in one built-in package java.lang.
5. To use String class in any other package there is no need to import string class explicitly. There are two types ways of creating String class Objects, a. Using new Operator.
   Ex: String rv = new String("hello");

b. By using double quotes.(without using new operator). Ex:
   String rv="Hello";

NOTE:

1. String variables are reference variables
2. String is a class and even it is also a "Sub class" to Object class.
3. All the non-static method of object class will be inherited to String class.
4. The 3 non-static methods of Object class, to_String(), equals() and hashcode() are overridden in String class based on the String values of the object.
5. to_String() is overridden in the String class to give the String value of the Object.
6. equals() has been overridden in the String class to return true if String values are same, otherwise false(case-sensitive).
7. hashcode() has been overridden in String class to give has# based in the String value of the Object.

Example:

public class Test1
{
public static void main(String[] args)
{
String rv1 = new String("Hello");
String rv2 = new String("hello");

```
            System.out.println(rv1);
            System.out.println(rv2);
            System.out.println(rv1.equals(rv2));
            System.out.println(rv1==rv2);
            System.out.println(rv1.hashCode());
            System.out.println(rv2.hashCode());
      }
}
```

Output: Hello
hello false false
69609650
99162322

Example:

```
public class Test2
{
      public static void main(String[] args)
      {
            String rv1="Hey";
            String rv2="Hey";



            System.out.println(rv1);
            System.out.println(rv2);
            System.out.println(rv1.equals(rv1));
            System.out.println(rv1.hashCode());
            System.out.println(rv2.hashCode());
      }
}
```

Output:
Hey
Hey true
72444
72444

Example:

```
public class Test3
{
      public static void main(String[] args)
      {
            String rv1 = new String("WateMelon");
            String rv2 = "WaterMelon";
            String rv3 = "Water";
            String rv4 = "Melon";
            String rv5 = "Water"+"Melon";
            String rv6 = rv3+"Melon"; // new Operator will be used.
            System.out.println(rv1==rv2);
```

```
        System.out.println(rv2==rv5);
        System.out.println(rv1==rv6);
    }
}
```

Output: false
true false

String Pool:

1. All the String objects will be stored in String pool of the heap memory.
2. String pool is further divided into 2 pools A) Constant pool:

    All the String Objects created by using "" (double quotes) will be stored in the constant pool.
    In constant pool, duplicates are not allowed.

Note: Duplicate means 2 string objects having same string values.

B) Non-Constant pool:

    String Objects which are created using new Operator will be stored in non-constant pool.
        In non-constant pool, duplicates are allowed.

Note: if reference variables are used for concatenation, then the resultant string will be created using new Operator.(so it will be stored in non-constant pool)

Ex: String rv1 = "water"; String rv2
    = rv1 + "Melon";

Here, the resultant String watermelon will be created using operator internally so the resultant String will be stored in on-constant pool

Example:

```
public class Test4
{
    public static void main(String[] args)
    {
        String rv1 = new String("WaterMelon");
        String rv2 = "WaterMelon";
        String rv3 = "Water";
        String rv4 = "Melon";
        String rv5 = "Water" + "Melon";
        String rv6 = rv3+"Melon";
        System.out.println(rv1.equals(rv2));
        System.out.println(rv2.equals(rv3));
        System.out.println(rv1.equals(rv6));
    }
}
```

Output: true
false true

NOTE: Always we should compare String using equals method, we should not use comparison operator to compare String value because comparison operator will compare the memory address of the String reference.

    So though String value are same, it returns "false".

equals() is best approach because it will compare the String value. String class is immutable.

Example:

```
public class Test5
{
	public static void main(String[] args)
	{
		String rv1 = "Subash";
		String rv2 = "Subash";
		String rv3 = "subash";

		System.out.println(rv1);
		System.out.println(rv2);
		System.out.println(rv3);
		System.out.println("------------")
;
		rv2="Chandra"; System.out.println(rv1);
		System.out.println(rv2);
		System.out.println(rv3);
		System.out.println("-------------");
		rv3="Bose";
		System.out.println(rv1);
		System.out.println(rv2);
		System.out.println(rv3);
	}
}
```

Output:

```
subash
------------
Subash Chandra
subash
-------------
Subash
Chandra Bose
```

Note:

String class is immutable. Justify?

Existing String Objects String value cannot be changed or modified nor updated. So we say string class as immutable.

String is a final class.

Example:

```
public class Test6
{
	public static void main(String[] args)
	{
		String rv = "I love India";
```

```
            System.out.println(rv.length());
            System.out.println(rv.charAt(3));
            System.out.println(rv.charAt(8));
            System.out.println("---------");
            System.out.println(rv.indexOf('v'));
            System.out.println(rv.indexOf('I'));
            System.out.println("---------------");
            System.out.println(rv.toLowerCase());
            System.out.println(rv.toUpperCase());
            System.out.println(rv.startsWith(" I love"));
            System.out.println(rv.endsWith("America"));
            System.out.println("------------");
            System.out.println(rv.contains("Love"));
            System.out.println("-------------");
            System.out.println(rv.substring(2));
            System.out.println(rv.substring(2, 9));
            System.out.println("----------");
            System.out.println(rv.isEmpty());
      }
}
```

Output:

```
12 o
n
---------
4
0
---------------
i love india I LOVE
INDIA
false false
-----------false
------------love
India love In
---------false
```

Example:

```
public class
Test7
{
      public static void main(String[] args)
      {
            String rv = " Welcome to World ";
            System.out.println(rv);
            System.out.println(rv.trim());
      }
}
```

Output:

Welcome to World Welcome to
World

Example: public class
Test8
{
public static void main(String[] args)
{
String rv = "AIDNI";
System.out.println("The original string is " +rv);
System.out.println("the reverse string is ::"); for(int
i=rv.length()-1;i>=0;i--)
{
System.out.print(rv.charAt(i));
}
}
}
Output: The original string is
AIDNI the reverse string is ::
INDIA

Example: public class
Test9
{
public static void main(String[] args)
{
String rv1 = "DAD"; String rv2 =
""; for(int i=rv1.length()-1;i>=0;i-
-)
{
rv2=rv2+rv1.charAt(i);
}
System.out.println("The original string :: " +rv1);
System.out.println("The reverse string :: " +rv2);
if(rv1.equalsIgnoreCase(rv2))
{
System.out.println(" Its a Palindrome");
}
else
{
System.out.println("its not a palindrome");
}
}
}
Output:
The original string :: DAD

```
The reverse string :: DAD Its a
 Palindrome
```

Example:

```
//count of total number of Occurrence of 'A'
public class Test10
{
      public static void main(String[] args)
      {
            String rv = "MALAYALAM"; int
            count =0;
            for(int i=0;i<rv.length();i++)
            { if(rv.charAt(i)=='A')
                  {
                        count++;
                  }
            }
            System.out.println("the count is " +count);
      }
}
```

Output:

```
the count is 4
```

Example:

```
public class Test11
{
      public static void main(String[] args)
      {
            String rv = "Google";
            System.out.println(rv); rv =
            rv.replace('G', 'D');
            System.out.println(rv);
      }
}
```

Output: Google
Doogle

26. ARRAYS

Arrays are contiguous memory allocation where-in we can store homogeneous element(similar kind of element) which will share the common name.
If you want to store the collection of similar data, then we can use arrays.

To store the huge collection of similar data, best approach is arrays because memory allocation will be continuous due to which processing will be faster.

Ex: If you want to store 100 integer data, best approach is arrays. If arrays are not used then we have to use 100 different variables.

NOTE: Arrays are treated as Objects in Java.

There are 2 types of Arrays:

1) Primitive Array:

The arrays in which we can store simple data like integer, fractional value etc., is called as Primitive array.

2) Object Array:

The array in which we can store object reference is called as Derived array or Object Array.

General Syntax for Declaration and Initialization:

type[] refvar = new type[size];

General Syntax to store element into Array:

refvar[index] = value;

Example:

rv = new int [5];

10	20	30	40	50
0	1	2	3	4

Primitive Array:

Example:

```
public class Test1
{
	public static void main(String[] args)
	{
		int[] rv = new int[5]; //Creating an integer array with 5 slots
		rv[0] = 10; rv[1] = 20; rv[2] = 30; rv[3] = 40; rv[4] = 50;
		System.out.println("length is:: " +rv.length);
		for(int i=0;i<rv.length;i++)
		{
			System.out.println(i);
		}
	}
}
```

Output:

```
length is::
5
0
1
2
3
4
```

Example:

```
public class Test2
{
	public static void main(String[] args)
	{
```

```
            int[] arr = new int[6];
            arr[0]=99; arr[1]=99;
            arr[2]=99; arr[3]=99;
            arr[4]=99; arr[5]=99; int
            total = 0;
            for(int i=0;i<arr.length;i++)
            {
                  System.out.println("marks in subject number " +(i+1)+ "::" +arr[i]);
                  total=total+arr[i];
            }
            System.out.println("----------------");
            System.out.println("total marks secured ::" +total);
            int avg = total/arr.length;
            System.out.println("------------------");
            System.out.println("Average marks is ::" +avg);
      }
}
```

Output: marks in subject number
1::99 marks in subject number
2::99 marks in subject number
3::99 marks in subject number
4::99 marks in subject number
5::99 marks in subject number
6::99
---
total marks secured ::594

Average marks is ::99

Example:

```
public class Test3
{
      public static void main(String[] args)
      {
            double[] darr = new double[5];
            darr[0]=10.1; darr[1]=10.2;
            darr[2]=10.3; darr[3]=10.4;
            darr[4]=10.5;
            System.out.println("---Enhanced for loop---"); for(double
            d:darr)
            {
                  System.out.println(d);
            }
            System.out.println("------------------");
            char[] carr = new char[5]; carr[0] =
            'A'; carr[1] = 'B';
```

```
            carr[2] = 'C'; carr[3] =
            'D'; carr[4] = 'E';
            for(char c:carr)
            {
                  System.out.println(c);
            }
      }
}
```

Output:

```
---Enhanced for loop--10.1
10.2 10.3 10.4
10.5
------------------
A B C
D
E
```

Example:

```
//To find largest element in a given integer array
public class Test4
{
      public static void main(String[] args)
      {
            int[] arr = new int[5]; arr[0] = 10;
            arr[1] = 20; arr[2] = 30; arr[3] =
            40; arr[4] = 50; int largest = arr[0];
            //Assumption
            for(int i=1;i<arr.length;i++)
            {
                  if(largest<arr[i])//Checking the assumption
                {
                        largest = arr[i];
                }
            }
            System.out.println("largest element is " +largest);
      }
}
```

Output:

```
largest element is 50
```

Example:

```
public class
Test5
{
      public static void main(String[] args)
      {
```

```
        int[] arr = new int[5]; arr[0] =
        50;
        arr[1] = 40; arr[2] = 30;
        arr[3] = 20; arr[4] =
        10; int index=0; int
        smallest = arr[0];
        for(int i=1;i<arr.length;i++)
        { if(smallest>arr[i])
            { smallest=arr[i];
                index=i;
            }
        }
        System.out.println("smallest element "+ smallest +" is available @index "+ index);
    }
}
```

Output:

smallest element 10 is available @index 4

Example:

```
//search given element in a array is existing or not, if its existing print the index of that element
otherwise print -1
public class Test6
{
    public static void main(String[] args)
    {
        int[] arr = new int[5];
        arr[0] = 10; arr[1] = 20;
        arr[2] = 30; arr[3] = 40;
        arr[4] = 50; int search =
        50;
        boolean b=true;
        for(int i=0;i<arr.length;i++)
        {
            if(search == arr[i])
            {
                b=false;
                System.out.println("element is available @ index " +i); break;
            }
        }
        if(b==true)
        {
            System.out.println("Element is not available " +(-1));
        }

    }
}
```

Output:

element is available @ index 4

```
//sort the elements of a given array in ascending order
public class Test7
{
	public static void main(String[] args)
	{
		int[] arr = new int[5]; arr[0]=50; arr[1]=40; arr[2]=30;
		arr[3]=20; arr[4]=10; System.out.println("Before");
		for(int i=0;i<arr.length;i++)
		{
					System.out.println(arr[i]);
		}
		for(int i=0;i<arr.length;i++)
		{ for(int j=i+1;j<arr.length;j++)
			{ if(arr[i]>arr[j])
				{
					int temp = arr[i]; arr[i] = arr[j];
							arr[j] = temp;
				}
			}
		}
		System.out.println("After");
		for(int i=0;i<arr.length;i++)
		{
					System.out.println(arr[i]);
		}
	}
}
```

Output: Before
50
40
30
20
10
After
10
20
30
40
50

```
//to find the sum of even number in a given integer array
public class Test8
{
	public static void main(String[] args)
	{
		int[] arr = new int[5]; arr[0]=10; arr[0]=11; arr[0]=12;
		arr[0]=13; arr[0]=14;
```

```
            int sum=0;
            for(int i=0;i<arr.length;i++)
            { if(arr[i]%2==0)
                  {
                                          sum=sum+arr[i];
                  }
            }
            System.out.println("elements sum is " +sum);
      }
}
```

Output: elements sum is 14

Derived Array:

Example:

```
class A
{
}
public class Test1
{
      public static void main(String[] args)
      {
            A[] ref = new A[5]; ref[0] = new A(); ref[1] = new A();
            ref[2] = new A(); ref[3] = new A(); ref[4] = new A();
            System.out.println("length is " +ref.length); for(int i=0;i<ref.length;i++)
            {
                                    System.out.println(ref[i]);
            }
      }
}
```

Output: length is 5 derivedarray.A@15db9742
derivedarray.A@6d06d69c derivedarray.A@7852e922
derivedarray.A@4e25154f derivedarray.A@70dea4e class B

```
{
      public String toString()
      {
            return "B class Object";
      }
}
public class Test2
{
      public static void main(String[] args)
      {
            B[] ref = new B[5];
            for(int i=0;i<ref.length;i++)
            {
                  ref[i]=new B();
```

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

Output:

```
B class Object
B class Object
B class Object
B class Object
B class Object
```

Example:

```
class Employee
{
      int empid; double salary;
      public Employee(int empid,double salary)
      {
            this.empid=empid;
            this.salary=salary;
      }
}
public class Test3
{
      public static void main(String[] args)
      {
            Employee[] ref=new Employee[5]; ref[0] = new Employee(10,100000);
            ref[1] = new Employee(20,200000); ref[2] = new Employee(30,300000);
            ref[3] = new Employee(40,400000); ref[4] = new Employee(50,500000);
            for(int i=0;i<ref.length;i++)
            {
                                                System.out.println(ref[i].empid + "::" + ref[i].salary);
            }
      }
}
```

Output:

```
10::100000.0
20::200000.0
30::300000.0
40::400000.0
50::500000.0
```

Example:

```
public class Test4
{
	public static void main(String[] args)
	{

		String[] ref = new String[5]; ref[0] =
		new String("Namaste"); ref[1] = new
		String("Subhodayam"); ref[2] = new
		String("Swagatam"); ref[3] = new
		String("Syankalam"); ref[4] = new
		String("Ratriam");
		for(int i=0;i<ref.length;i++)
		{
			System.out.println(ref[i].toUpperCase());
		}
	}
}
```

Output:

```
NAMASTE
SUBHODAYAM
SWAGATAM
SYANKALAM RATRIAM
```

Example:

```
public class Test5
{
	public static void main(String[] args)
	{
		String[] ref = new String[5];
		ref[0] = "AB"; ref[1] = "Cd";
		ref[2] = "EF"; ref[3] = "GH";
		ref[4] = "IJ"; for(int
		i=0;i<ref.length;i++)
		{ ref[i]=ref[i].toUpperCase();
		}
		Arrays.sort(ref);
		for(String s:ref)
		{
			System.out.println(s);
```

```
            }
            System.out.println(Arrays.binarySearch(ref, "AB"));
            System.out.println(Arrays.binarySearch(ref, "AB"));
            System.out.println(Arrays.binarySearch(ref, "ABCD"));
      }
}
```

Output:

```
AB
CD
EF
GH
IJ
0
0
-2
```

Example:

```
import java.util.Arrays;
public class Test6
{
      public static void main(String[] args)
      {
            int[] arr = new int[5];
            arr[0] = 50; arr[1] = 40;
            arr[2] = 30; arr[3] = 20;
            arr[4] = 10;
            Arrays.sort(arr);
            System.out.println(Arrays.binarySearch(arr,40));
            System.out.println(Arrays.binarySearch(arr,6));
            System.out.println(Arrays.binarySearch(arr,15));
            System.out.println(Arrays.binarySearch(arr,45));
      }
}
```

Output:

```
3
-1
-2
-5
```

Example:

```
public class Test7
{
      public static void main(String[] args)
      {
            String rv="We live in Earth"; String[]
            ref=rv.split(" ");
```

```
            for(String s:ref)
            {
                  System.out.println(s);
            }
            System.out.println("length of the String is " +ref.length);
      }
}
```

Output:

We live in Earth length of
the String is 4 **Example:**

```
public class Test8
{
      public static void main(String[] args)
      {
            String rv="Customer Relationship Management";
            String[] ref=rv.split(" ");
            String a = " ";
            a = a+ref[0].charAt(0)+ref[1].charAt(0)+ref[2].charAt(0);
            System.out.println(a);
      }
}
```

Output:

CRM

Example:

```
public class Test9
{
      public static void main(String[] args)
      {
            String rv = "Customer Relationship Management, Enterprise Relationship Planning";
            String[] ref = rv.split(" "); String a
            = " ";
            for(int i=0;i<ref.length;i++)
            { a=a+ref[i].charAt(0);
            }
            System.out.println(a);
      }
}
```

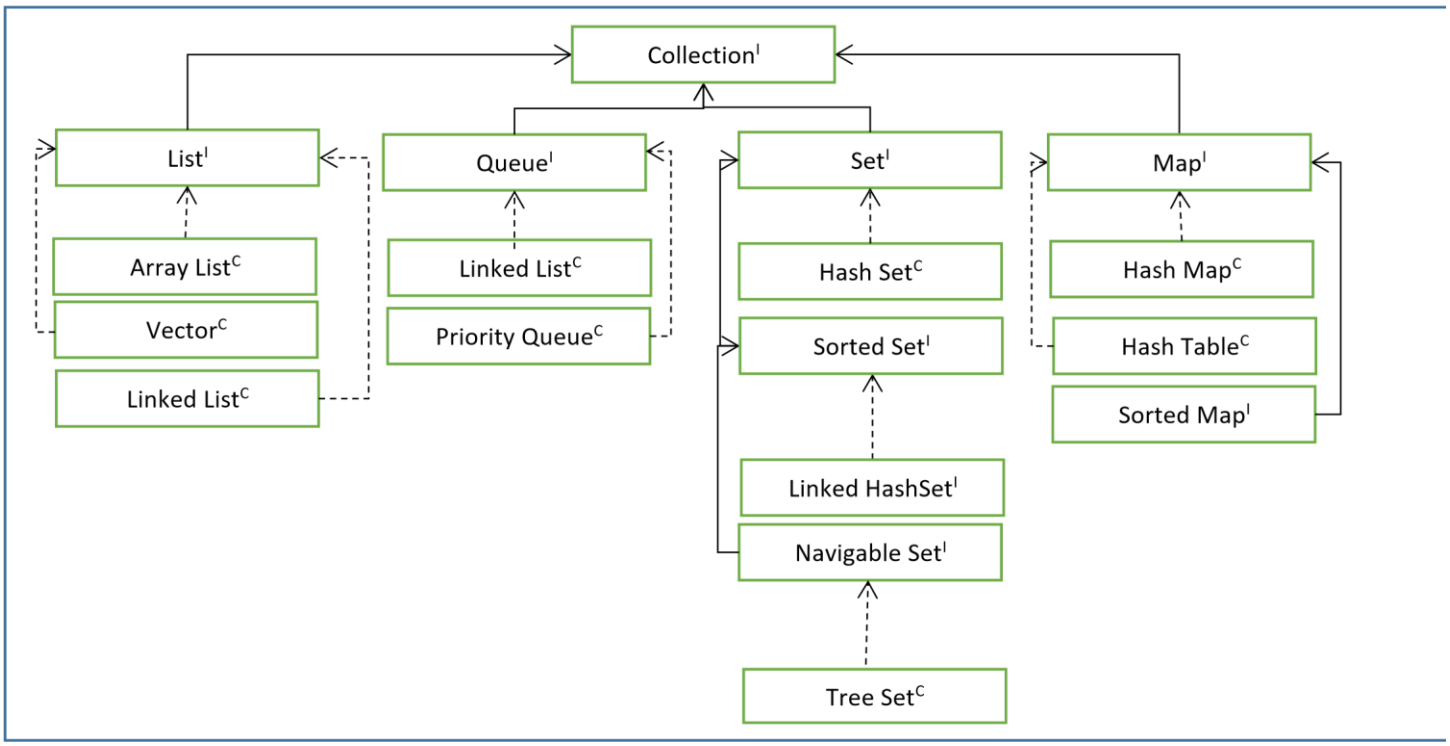
Output:

CRMERP

Limitations of Arrays:

1. Size is fixed i.e., once an array is created with some size it cannot be expanded or compressed i.e., arrays size in not flexible.
2. Some times we may waste the memory or sometimes we may need additional memory.
3. In arrays we can store only homogeneous data or elements.

27. COLLECTION



4. There are very less built-in functionality in array.

Collections is a framework or API, in which there are lot of classes, interfaces and built-in functionality which will serve the common purpose.

1. All the collections framework functionality has been implemented using standard data structure like(stack, queue, single and doubly list).
2. All the collection framework related class and interfaces are available in java.util package.
3. In collections Size will be dynamic and we can store heterogeneous elements.
4. Collections are called as "Dynamic Array".(All the elements added in collections will be stored as an Object)
5. Collections is a root interface in Collections framework which defines some common features.
6. Collections is a concrete class in collections framework which provides some utility method like sorting, searching , etc..,.
7. Base on the variations in the requirements, collection is categorized into 4 types. **a. List**
   b. Queue
   c. Set
   d. Map

a. List:

1. To maintain the insertion order with indexing we should go for list tyoe of collections.
2. Features of List:

a) List is dynamic
b) In list, we can store heterogeneous data.
c) List is a index type of collection i.e., the elements will be stored upon index with auto-indexing and sequential indexing.
d) List allows duplicate.
e) List allows Null.

f) As it is a index based collection to fetch the elements we can use index.

List is a Interface inheriting from another interface collection There
are 3 classes implementing the list interface:

1. ArrayList
2. Vector
3. LinkedList.

1. ArrayList:

- ✓ Arraylist memory allocation is contiguous, so random access to any element will be faster.
- ✓ But insertion operation is not good in arraylist because shuffling should happen to rest of element.
- ✓ Whenever requirement is more on searching operation or random access to any element required is more then we should choose ArrayList.

2. VectorList:

- ✓ Vector class methods are synchronized methods or thread safe methods i.e., when one thread is executing a vector method another Thread cannot execute the vector method.
- ✓ Ex: BackButton implementation will be done using _____ to record the navifgation of user in web page.

3. LinkedList:

- ✓ LinkedList memory allocation is not contiguous i.e., linkedlist ,memory allocation is exactly like linkedlist data structure i.e., one element will have linked to the another element.

.

- ✓ LinkedList elements will be scattered in the memory.
- ✓ Insertion operation will be good in LinkedList because one link will be attached to new element which is inserted.
- ✓ While coding if insertion operation is more like updating record etc., then we should choose linkedList.
- ✓ Random access to any element is not good in LinkedLIst.

Example:

```
import java.util.ArrayList;
class A
{

}
public class Test1
{
    public static void main(String[] args)
    {
        ArrayList list = new ArrayList();
        list.add(10); list.add(10.5);
        list.add('A'); list.add("hey");
        list.add(true); list.add(10);
        list.add(new A());
        System.out.println("list is " +list);
```

```
            System.out.println("size of array list is " +list.size());
            for(int i=0;i<list.size();i++)
            {
                  System.out.println(list.get(i)); }
       }
list is [10, 10.5, A, hey, true, 10, collection.A@15db9742] size
of array list is 7
10
10.5
A hey true 10
collection.A@15db9742
```

Example:

```
import java.util.ArrayList;
public class Test2
{
      public static void main(String[] args)
      {
            ArrayList list = new ArrayList();
            list.add(10); list.add(10.5);
            System.out.println(list);
            list.add("Hi");//Appends the given element to the end System.out.println(list);
            list.add(1,90);//Inserts the given element into the        given index
            System.out.println(list);
            list.set(0, true);//modifies the given indexed element with new Element
            System.out.println(list);
      }
}
```

Output:

```
[10, 10.5]
[10, 10.5, Hi]
[10, 90, 10.5, Hi] [true, 90,
10.5, Hi]
```

Example:

```
import java.util.ArrayList;
public class Test3
{
      public static void main(String[] args)
      {
            ArrayList list1 = new ArrayList();
            ArrayList list2 = new ArrayList(); ArrayList
            list3 = new ArrayList();
            list1.add(10); list1.add(20.5);
            list1.add('A'); list2.add(30.3);
            list2.add("hi"); list2.add(90);
```

```
            list3.add('C'); list3.add(true);
            list3.add(50);

            System.out.println(list1);
            System.out.println(list2);
            System.out.println(list3);
            list1.addAll(list2); System.out.println(list1);
            list2.addAll(list3); System.out.println(list2);
      }
}
```

Output:

```
[10, 20.5, A]
[30.3, hi, 90]
[C, true, 50]
[10, 20.5, A, 30.3, hi, 90] [30.3,
hi, 90, C, true, 50]
```

Example:

```
import java.util.ArrayList;
public class Test4
{
      public static void main(String[] args)
      {
            ArrayList list=new ArrayList();
            list.add("Bendakaluru");
            list.add("Guwahati");
            list.add("Bhubaneswar");
            list.add("Pune");
            System.out.println(list);
            list.remove(0);
            System.out.println(list);
            list.remove("Pune");
            System.out.println(list); list.clear();
            System.out.println(list);
      }
}
```

Output:

```
[Bendakaluru, Guwahati, Bhubaneswar, Pune]
[Guwahati, Bhubaneswar, Pune]
[Guwahati, Bhubaneswar]
[]
```

Example:

```
import java.util.ArrayList;
public class Test5
{
      public static void main(String[] args)
```

```
    {
        ArrayList list=new ArrayList();
        list.add("Manual"); list.add("SQL");
        list.add("Java"); list.add("Selenium");
        list.add("AJava");
        System.out.println(list.remove(0));
        System.out.println(list);
        System.out.println(list.remove("SQL"));
        System.out.println(list); }
```

```
Manual
[SQL, Java, Selenium, AJava]
true [Java, Selenium, AJava]
```

Example:

```
import java.util.ArrayList;
public class Test6
{
    public static void main(String[] args)
    {
            ArrayList list1 = new ArrayList();
        ArrayList list2 = new ArrayList();
        ArrayList list3 = new ArrayList();
        list1.add(10); list1.add(20);
        list1.add(30); list1.add(40);
        list1.add(50); list2.add(40);
        list2.add(50); list2.add(60);
        list2.add(70); list2.add(80);
        list3.add(60); list3.add(70);
        list3.add(80); list3.add(90);
        list3.add(100);
        System.out.println(list1);
        System.out.println(list2);
        System.out.println(list3); list1.removeAll(list2);
        System.out.println(list1); list2.retainAll(list3);
        System.out.println(list2);
    }
}
```

Output:

```
[10, 20, 30, 40, 50]
[40, 50, 60, 70, 80]
[60, 70, 80, 90, 100]
[10, 20, 30]
[60, 70, 80]
```

Example:

```
import java.util.ArrayList;
```

```
public class Test7 extends ArrayList
{
    public static void main(String[] args)
    {
        Test7 list = new Test7();
        list.add(10); list.add(20);
        list.add(30); list.add(40);
        list.add(50); list.add(60);
        list.add(70); list.add(80);
        list.add(90); list.add(100);
        System.out.println(list);
        list.removeRange(2, 7);
        System.out.println(list);
    }
}
```

Output:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100] [10,
20, 80, 90, 100]
```

Example:

```
import java.util.ArrayList; import
java.util.Collections;
public class Test8
{
    public static void main(String[] args)
    {
        ArrayList list =new ArrayList();
        list.add("Encapsulation");
        list.add("TypeCasting"); list.add("Abstraction");
        list.add("WrapperCLass"); list.add("Collections");
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Output:

```
[Encapsulation, TypeCasting, Abstraction, WrapperCLass, Collections]
[Abstraction, Collections, Encapsulation, TypeCasting, WrapperCLass]
```

Example:

```
import java.util.ArrayList; import
java.util.Collections;
public class Test9
{
    public static void main(String[] args)
    {
```

```
        ArrayList list =new ArrayList();
        list.add("Encapsulation"); list.add("TypeCasting");
        list.add("Abstraction"); list.add("WrapperCLass");
        list.add("Collections");
        System.out.println("Before: " +list);
        Collections.sort(list);
        System.out.println("After:" +list);
        Collections.reverse(list);
        System.out.println("reverse of sort:" +list);
    }
```

Before: [Encapsulation, TypeCasting, Abstraction, WrapperCLass, Collections]
After:[Abstraction, Collections, Encapsulation, TypeCasting, WrapperCLass] reverse of sort:[WrapperCLass, TypeCasting, Encapsulation, Collections, Abstraction]

Example:

```
import java.util.ArrayList; import
java.util.Collections;
public class Test10
{
    public static void main(String[] args)
    {
        ArrayList list = new ArrayList();
        list.add('E'); list.add('D');
        list.add('C'); list.add('B');
        list.add('A');
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
        System.out.println(Collections.binarySearch(list, 'A'));
        System.out.println(Collections.binarySearch(list, 'E'));
    }
}
```

Output:

```
[E, D, C, B, A] [A, B,
C, D, E]
0
4
```

Example:

```
import java.util.ArrayList; import
java.util.Collections;
public class Test11
{
    public static void main(String[] args)
    {
```

```
            ArrayList list = new ArrayList();
            list.add("Mango");
            list.add("Apple");
            list.add("Gauva"); list.add(90);
            list.add("Orange");
            list.add("Papaya");
            Collections.sort(list);
            System.out.println(list);
        }
}
```

Output:

java.lang.String cannot be cast to java.lang.Integer

Example:

```
import java.util.ArrayList; import
annotations.override; class Z
{
	@override public String
	toString()
	{
		return "Z class Object";
	}
}
public class Test12
{
	public static void main(String[] args)
	{
		ArrayList list = new ArrayList();
		list.add(new Z()); list.add(new
		String("Hello")); list.add("hi");
		list.add(new Integer(10));
		list.add(105); list.add(true);
		list.add('A'); for(Object obj:list)
		{
			System.out.println(obj);
		}
	}
}
```

Output:

```
Z class Object Hello
hi
10 105 true
A
```

Example:

```
import java.util.ArrayList;
public class Test13
{
	public static void main(String[] args)
	{
		ArrayList list = new ArrayList();
		list.add("Bendakaluru");
		list.add("BBSR");
		list.add("Magaluru");
		list.add("Hyderabad"); String
		s1=(String) list.get(0);
		System.out.println(s1.toUpperCase());

		String s2=(String) list.get(1);
```

```
            System.out.println(s1.length());

            String s3=(String) list.get(2);
            System.out.println(s1.charAt(3));
            String s4=(String) list.get(3);
            System.out.println(s4.indexOf('y'));
      }
}
```

Output:

```
BENDAKALURU 11 d
1
```

Example:

```
import java.util.ArrayList;
class Mobile
{
      void Message()
      {
            System.out.println("Message method");
      }
}
public class Test14
{
      public static void main(String[] args)
      {
            ArrayList list = new ArrayList();
            list.add(new Mobile());
            list.add(new Mobile());
            list.add(new Mobile()); Mobile
            m1=(Mobile)list.get(0);
            m1.Message();
            Mobile m2=(Mobile)list.get(1); m2.Message();
            Mobile m3=(Mobile)list.get(2);
            m3.Message();
      }
}
```

Output:

```
Message method
Message method Message
method
```

b. Queue:

1.To maintain the General Queue, we go for Queue type of collection.

2.Features of Queue are:

a) It is dynamic.

b) We can store heterogeneous element.

c) Queue is not a indexed type of collections i.e., elements will not be stored upon index.
d) Null is not allowed in Queue.
e) Duplicates are allowed in Queue.
f) As Queue is not a indexed type of collection we cannot fetch element using index.
g) We should use 2 methods - peak() & poll()

Queue is a interface inheriting from another interface, Collection.
There are 2 classes implementing queue interface feature 1. Priority
Queue.
2. LinkedList.

Example:

```
import java.util.PriorityQueue;
public class Test1
{
    public static void main(String[] args)
    {
        PriorityQueue queue = new PriorityQueue();
        queue.add(50); queue.add(40);
        queue.add(30); queue.add(20);
        queue.add(10);
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue);
        System.out.println(queue.poll());
    }
}
```

Output:

```
[10, 20, 40, 50, 30]
10
[20, 30, 40, 50]
20
[30, 50, 40]
30
[40, 50]
40
[50] 50
```

Example:

```
import java.util.PriorityQueue;
public class Test2
{
```

```
public static void main(String[] args)
    {
        PriorityQueue queue = new PriorityQueue();
        queue.add(10.5); queue.add(10.4);
        queue.add(10.3); queue.add(10.2);
        queue.add(10.1);
        System.out.println(queue);
        System.out.println(queue.peek());
        System.out.println(queue);
        System.out.println(queue.peek());
        System.out.println(queue);
    }
}
```

Output:

```
[10.1, 10.2, 10.4, 10.5, 10.3]
10.1
[10.1, 10.2, 10.4, 10.5, 10.3]
10.1
[10.1, 10.2, 10.4, 10.5, 10.3]
```

Example:

```
import java.util.PriorityQueue;
public class Test3
{
    public static void main(String[] args)
    {
        PriorityQueue queue = new PriorityQueue();
        queue.add('E');
        queue.add('D');
        queue.add('C');
        queue.add('B');
        queue.add('A'); int
        size=queue.size();
        for(int i=1;i<=size;i++)
        {
            System.out.println(queue.poll());
        }
    }
}
```

Output:

```
A B C
D
E
```

Example:

```
import java.util.PriorityQueue;
public class Test4
{
	public static void main(String[] args)
	{
		PriorityQueue queue = new PriorityQueue();
		queue.add("Mango"); queue.add("Mango");
		queue.add("70"); queue.add("Mango");
		queue.add("$@"); System.out.println(queue);
	}
}
```

Output:

```
[$@, 70, Mango, Mango, Mango
```

C. Set:

1. If you want to store the unique element by avoiding duplicate then we should choose set type of collection.
2. Features of Set:

a) It is dynamic.
b) We can store heterogeneous element.
c) Duplicates are not allowed.
d) It is not a index based collection.
e) Null is allowed.
f) As it is not a index based collection we cannot fetch the elements using index.

Set is a interface inheriting from another interface collection.

There is 3 classes implementing set:

1. HashSet
2. LinkedList
3. TreeSet

Example:

```
import java.util.HashSet;
public class Test1
{
	public static void main(String[] args)
	{
		HashSet set = new HashSet();
		set.add(10); set.add(10);
		set.add(20.5); set.add(30);
		set.add('A'); set.add(10);
		set.add(true); set.add(null);
		System.out.println(set.size());
		System.out.println(set);
	}
}
```

Output:

```
6
```

[null, A, 20.5, 10, 30, true]

Example:

```
import java.util.LinkedHashSet; public
class Test2
{
	public static void main(String[] args)
	{
		LinkedHashSet set = new LinkedHashSet();
		set.add(10); set.add(10);
		set.add(20.5); set.add('A');
		set.add(10); set.add(true);
		set.add(null);
		System.out.println(set.size());
		System.out.println(set);
	}
}
```

Output:

5 [10, 20.5, A, true, null]

Example:

```
import java.util.TreeSet;
public class Test3
{
	public static void main(String[] args)
	{
		TreeSet set = new TreeSet();
		set.add(10); set.add(30);
		set.add(20); set.add(40);
		set.add(10); set.add(20);
		set.add(60); set.add(50);
		System.out.println(set.size());
		System.out.println(set);
	}
}
```

Output:

6

[10, 20, 30, 40, 50, 60]

Example:

```
import java.util.HashSet;
public class Test4
{
	public static void main(String[] args)
	{
		HashSet set = new HashSet();
		System.out.println(set.add(10));
```

```
            System.out.println(set.add(10));
            System.out.println(set.add(20.5));
            System.out.println(set.add('A'));
            System.out.println(set.add(20.5)); System.out.println(set.add("hi"));
            System.out.println(set.add(null));
            System.out.println(set.add(20.5));
      }
}
```

Output: true
false true true
false true true
False **NOTE:**

1.If you want to store unique elements then we should choose “Set” type of Element.
2.If you want to store unique and not expecting any insertion order we should use “HashSet “.
3.If you want to store unique elements b maintaining insertion order, use “LinkedHashset”.
4.If you want to store unique elements with auto sorting then choose “TreeSet” class 5.To store the elements in List, set, Queue we should use add method.

add() will perform 2 Operation:
1- It will store the given element.
2- It will return Boolean value true if it has stored the given element otherwise false.

6. add() return type is Boolean, based on the add() return value we can be notifies about whether the given element is stored or not.

Example:

```
import java.util.HashSet; import
java.util.Iterator; import
java.util.LinkedHashSet; public class
Test5
{
      public static void main(String[] args)
      {
            HashSet set = new LinkedHashSet();
            set.add(10); set.add(20.5);
            set.add('A'); set.add("Hi");
            set.add(90);
            System.out.println(set); Iterator itr
            = set.iterator();
            while(itr.hasNext());
            {
                  System.out.println(itr.next()); itr.remove();
            }
            System.out.println(set);
      }
}
```

Output: [10, 20.5, A,
Hi, 90]

Example:

```
import java.util.HashSet;
public class Test6
{
      public static void main(String[] args)
      {
            HashSet set = new HashSet();
            set.add(10); set.add(10);
            set.add(10);

}
```

Output: true
false true true
false true true
true
6
[null, A, Hi, 20.5, 20.3, 10]

Example:

```
import java.util.HashSet; import
java.util.Iterator;
public class Test6
{
      public static void main(String[] args)
      {
            HashSet set = new HashSet();
            set.add(10); set.add(10.5);
            set.add(true); set.add(null);
            set.add('A'); set.add("hi");
            System.out.println(set); Iterator itr
            = set.iterator();
            while(itr.hasNext());
            {
                  System.out.println(itr.next());
            }
            System.out.println(set); Iterator
            itr2 = set.iterator();
            while(itr2.hasNext());
            {
                  System.out.println(itr2.next());
            }
            System.out.println(set);
      }
}
```

Output:

[null, A, hi, 10.5, 10, true]

NOTE:

1. to fetch the element "Set" type of collection we should use iterator method.
2. Iterator is a interface, which is having 3 abstract method implemented in some class
   a.HashNext();
   b.Next();
   c.Remove();

a. HashNext():
   - It is implemented like a pointer. Initially it will be pointing to first element of set. HashNext() will return a Boolean Value true if it us pointing to any element, if it is not pointing to any element i.e., if its pointing to null then it will return a Boolean value null. - HashNext() return type is Boolean.

b. Next():
   Used to fetch the elements.
   Next will perform 2 operations
   - It will return element which is pointed by HashNext.
   - It will increment the HashNext pointer element to Next method. -Next return type is Object.

c. Remove():
   - Used to remove the elements from set while iterating.
   - Removes the element which is returned by Next(). - Removes return type is void.

Example:

```
import java.util.LinkedHashSet;
public class Test7
{
    public static void main(String[] args)
    {
        String rv = "aaabbbccc";
        LinkedHashSet set = new LinkedHashSet(); for(int
        i=0;i<rv.length();i++)
        { set.add(rv.charAt(i));
        }
        System.out.println(set);
    }
}
```

Output: [a, b, c]

Example:

```
import java.util.Iterator; import
java.util.LinkedHashSet;
public class Test8
{
    public static void main(String[] args)
    {
        LinkedHashSet set = new LinkedHashSet();
```

```
            set.add("Abhay");
            set.add("Disha");
            set.add("Nisha");
            set.add("Jisha"); Iterator itr =
            set.iterator();
            while(itr.hasNext())
            {
                  String s = (String)itr.next();
                System.out.println(s.toUpperCase());
            }
      }
}
```

Output: ABHAY
DISHA NISHA
JISHA

Example:

```
import java.util.Iterator; import
java.util.TreeSet;
public class Test9
{
      public static void main(String[] args)
      {
            TreeSet set = new TreeSet();
            set.add("einsein"); set.add("Dravid");
            set.add("Chetak"); set.add("Tata");
            System.out.println(set);
            Iterator itr = set.iterator();
            String s1=(String)itr.next();
            System.out.println(s1.toUpperCase());

            String s2=(String)itr.next();
            System.out.println(s1.length());

            String s3=(String)itr.next();
            System.out.println(s1.charAt(3));

            String s4=(String)itr.next();
            System.out.println(s4.indexOf(1));
      }
}
```

Output:

[Chetak, Dravid, Tata, einsein]
CHETAK 6
t -1

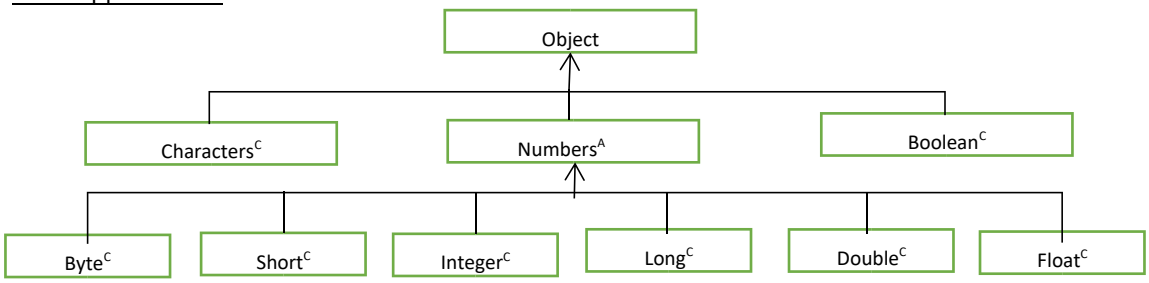
28. WRAPPER CLASS

Each of the primitives has a corresponding class that provides several useful abilities that are otherwise unavailable to a primitive.

Just think wrapper classes as each being a package of useful abilities wrapped around a primitive of the corresponding type.

Primitive	Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Note: String is an object rather than a primitive) Hierarchy of Wrapper Class:



Boxing:

Wrapping up the primitive data into an object by using wrapper classes is called as Boxing Operation. or

Converting the primitive data into Object by using wrapper class is called as Boxing Operation.

1.The primitive data types are not objects they do not belong to any class, they are defined in language itself.

2.Sometimes, it is required to convert data types into objects in Java Program.

3.By using operation we can design Generic functions which can receive any input

4.Through Boxing we can convert everything into objects.

5.To achieve boxing operation, there are some Wrapper class provided.

6.All the wrapper classes are concrete, final and immutable.

7.All the wrapper classes are available in java.lang.package.

8.All the 6 number related wrapper class inherits from a class called NUMBER.

9.Number is an abstract class of java.lang.package and Number is a sub class to object class.

10.All the wrapper class provides some functions which performs operation on the related data. Ex: in character wrapper class there is a static method which will convert the given character value to upper case.

11. In Integer wrapper class there is a method called parseInt which will convert the Integer value which in the String form into its Primitive int form. Ex: int a = Integer.parseInt("100");

Note:

1. All the wrapper class are sub-class to “object class”.
2. Whenever we print reference variable of Wrapper classes we will get the Boxed data in the String form.

Example: public

```
class A
{
    public static void main(String[] args)
    {
        Integer i1=new Integer(10);//Boxing Operation
        Double d1=new Double(10);//Boxing Operation
        System.out.println(i1);
        System.out.println(d1);
    }
}
```

Output:

```
10
10.0
```

Example: public

```
class B
{
    public static void main(String[] args)
    {
        Integer i1=new Integer(10); //Boxing Operation int
        a=i1.intValue();//Un-Boxing Operation
        System.out.println("value of a is " +a);

        Double d1=new Double(10.5);//Un-Boxing Operation double
        d=d1.doubleValue();//Un-Boxing Operation
        System.out.println("Value of d is " +d);
    }
}
```

Output: value of a is
10 Value of d is 10.5

Example: public

```
class C
{
    public static void main(String[] args)
    {
        Integer i1=90; //Auto-Boxing int
        i=i1; System.out.println(i);

        Character ch1='A'; //Auto-Boxing char
        ch=ch1; //Auto-UnBoxing
        System.out.println(ch);
```

```
        Boolean b1=true;//Auto-Boxing boolean
        b=b1;//Auto-UnBoxing
        System.out.println(b);
    }
}
```

Output:

```
90 A
true
```

Example:

```
class X
{

} public class D
{ static void fun(Object obj)
    {
        System.out.println("done");
    }
    public static void main(String[] args)
    { fun(new A()); fun(new String("Hello"));
        fun("hi"); fun(new Integer(10));
        fun(90.5); fun('A'); fun(true);
    }
}
```

Output:

```
done
done done done
done done done
```

Example:

```
public class E
{ public static void main(String[] args)
    {
        System.out.println((Integer.toBinaryString(10)));
        System.out.println((Character.toUpperCase('a')));
        System.out.println((Character.isUpperCase('A')));
        System.out.println((Character.isDigit(9)));
        System.out.println((Character.isAlphabetic('A')));
    }
}
```

Output:

```
1010 A true
false true
```

Example:

```
public class F
{ public static void main(String[] args)
    { int a=Integer.parseInt("100");
        System.out.println(a);
        System.out.println(a);
```

```
        String s1="100";
        String s2="200"; System.out.println(s1+s2);

        int b=Integer.parseInt(s1); int
        c=Integer.parseInt(s2);
        System.out.println(b+c);
    }
}
```

Output:

100
100
100200 300

UnBoxing:

Getting the boxed data in its primitive form with the help of some value methods of wrapper classes is called as UnBoxing. or
Converting the Primitive data which is boxed as an object into its Primitive form by using value methods of wrapper classes is called as UnBoxing.

To perform the primitive operations on the boxed data we will go for UnBoxing operation.

Note: Both Boxing and UnBoxing is Automatic, which is also called as Auto-Boxing and Auto-UnBoxing. Ex: Integer rv1=10

Int a=rv1;

Is Java 100% OOP language?

No, any language which will not support Primitive types but deals everything in the form of objects, then such programming language are called Pure or 100% Object Oriented Language. As Java supports Primitive type also, it is not 100% Object Oriented.

Note: Small-talk is 100% OOP language. Java, C++, .net are not 100% OOP language.

29. EXCEPTION HANDLING

Exception are run-time errors some statements will be syntactically correct but behave abnormally at run time, then we get run-time errors which is called as Exception.

Whenever we get run-time errors, program will stop its execution. In order to continue the program executions we should handle run-time errors by using try-catch Blocks.

Definition: (Exception)

Whenever any java statement produces an abnormal condition at tun-time, JVM will create any one of the appropriate Throw-able class type object and it will throe it to the method in which an abnormal statement is developed, this is called as Exception and handling those exception using try-catch block is called as Exception Handling.

An exception can occur for many different reasons, including the following:

1. A user has entered invalid data.
2. A file that needs to be opened cannot be found.

3. A network connection has been lost in the middle of communications or the JVM has run out of memory.

Handling run-time errors using try-catch block is called as Exception Handling.

Risky statement should be developed inside try block and a catch block should be written to address the exception.

The three categories of exceptions:

1. **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer.

Example: if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

2. **Run-time exceptions:** A run-time exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, run-time exceptions are ignored at the time of compilation.

3. **Errors:**These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error.

Example: if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Example: public

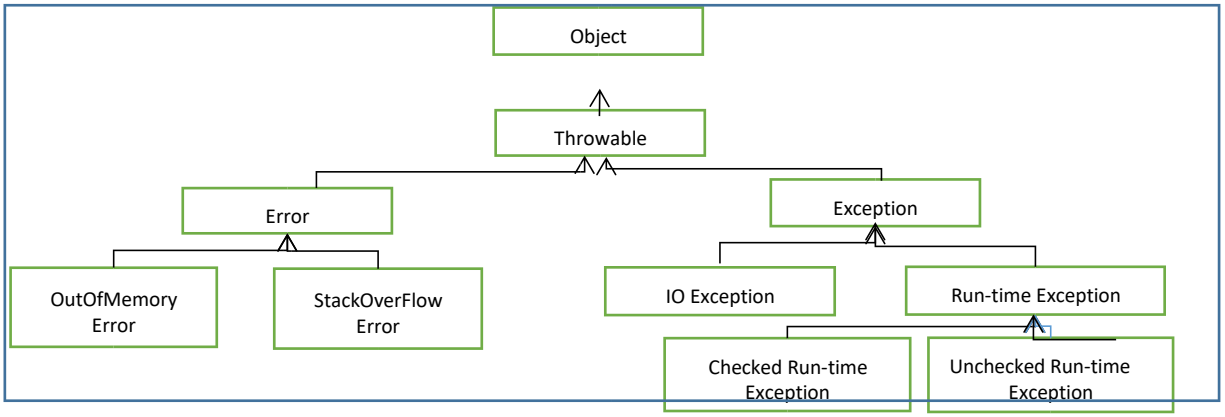
```
class A
{
    public static void main(String[] args)
    { try
        {
            int a=Integer.parseInt("hey");
        }
        catch (NumberFormatException e)
        {
            System.out.println("caught");
        }
    }
}
```

Output: caught

B

```
        try
        {
            int a=10/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught");
        }
    }
}
```

Output: caught



Java Unchecked Run-time Exception.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array elementof an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.

SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationExcepiton	An unsupported operation was encountered.

Java Checked Exceptions:

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implementthe Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Example:

```
public class C
{
    public static void main(String[] args)
    {
        int[] arr={10,20}; try
        {
            System.out.println(arr[900]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Output: 900

Exceptions Methods: Following is the list of important methods available in the Throwable class.

SN Methods with Description

1. **public String getMessage():** It is a non-static method which will be available in every exception Object. It **r**eturns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2. **public Throwable getCause():** Returns the cause of the exception as represented by a Throwable object.
3. **public String toString():** Returns the name of the class concatenated with the result of getMessage()
4. **public void printStackTrace():** Prints the result of toString() along with the stack trace to System.err, the error output stream.

5 **public StackTraceElement [] getStackTrace():** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

6 **public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Note:

Throw-able class is the super-most class for the Exception related class.
Throw-able class is the concrete class of java.lang.package Throw-able class
is a sub class to object class.

D

```
        int a=10; int b=0; try
        {
            int result=a/b;
            System.out.println("the result is " +result);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

/ by zero

Example: class Z

```
{

}
public class E
{
    public static void main(String[] args)
    {
    // Z z1= null; Z z1=new Z();
        try
        {
            System.out.println(z1.hashCode());
        }
        catch(ArithmeticException e)
        {
                System.out.println("caught!");
        } finally
        {
            System.out.println("Runnig Finally block ¥n");
```

```
            }
        }
}
```

Output:

366712642
Runnig Finally block

```
class Z
{

}
public class E
{
	public static void main(String[] args)
	{
		Z z1= null;
	//	Z z1=new Z(); try
		{
			System.out.println(z1.hashCode());
		}
		catch(ArithmeticException e)
		{
			System.out.println("caught!");
		} finally
		{
			System.out.println("Runnig Finally block ¥n");
		}
	}
}
```

Output:

```
Runnig Finally block
Exception in thread "main" java.lang.NullPointerException
```

Example:

```
public
class F
{
	public static void main(String[] args)
	{ try
		{
			int a=10/0;
		}
		catch(NumberFormatException e)
		{
			System.out.println("Runnig 1st catch");
		}
		catch(ArithmeticException e)
		{
			System.out.println("Running 2nd catch");
		}
	}
}
```

Output:

```
Running 2nd
catch
```

Note:

1 try block can be followed by multiple catch block but only appropriate catch block will be executed.

G

```
        int[] arr1={10,20,30,40,50,60,70,80}; int[]
        arr2={10,0,30,0,50};
        for(int i=0;i<arr1.length;i++)
        { try
            {
                System.out.println(arr1[i]/arr2[i]);
            }
            catch(ArithmeticException e)
            {
                System.out.println(e.getMessage());
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("No such index");
            }
        }
    }
}
```

Output: 1

```
/ by zero
1
/ by zero
1
No such index
No such index
No such index
```

Example: public

```
class H
{
    public static void main(String[] args)
    { try
        {
            int[] arr=new int[-10];
        }
        catch(Throwable e)
        {
            System.out.println(e.getMessage());
            System.out.println("Caught");
        }
    }
```

```
}
```

Output: null
Caught

Note: Though throw-able catch block can handle any type of exception we should not use throw-able catch block alone i.e., while handling exceptions we should follow an order while writing catch block. We should write the catch block from most specific exception class to most Generic Exception class.

```
public class I
{
    public          main(String[] args)
    { try
        {
            int a =10/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage());
            System.out.println("1st caught");
        }
        catch(RuntimeException e)
        {
            System.out.println(e.getMessage());
            System.out.println("2nd caught");
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
            System.out.println("2nd caught");
        }
        catch(Throwable e)
        {
            System.out.println(e.getMessage());
            System.out.println("2nd caught");
        }
    }
}
```

Output:

/ by zero
1st caught

There are 2 types of EXCEPTIONS, 1.

Unchecked Exceptions:

The exceptions which cannot be identified by compiler is called as “Unchecked Exceptions”. All the exceptions related classes which are sub-class to run-time exception class and error class will fall under Unchecked Exceptions.

Ex: ArithemeticException, NumberformatException etc,

Ex: int result = a/b;
Here compiler cannot identify that certain statements might generate a abnormal condition at run-time.

2. Checked Exceptions:

Checked Exceptions will be detected by compiler. All the sub-classes which are sub class to all the exceptions class will fall under Checked Exceptions category.
Ex: IOException, fFleNotFoundException, InterupptedException.

Here compiler can identify that certain statements might generate a abnormal condition at run-time.

If Compiler identifies any checked Exceptions, then programmer should handle it,
There are 2 ways to handle checked Exceptions,

A. try-catch block
B. Using throws keyword.

```
class M
{
                    test() throws Exception
    {
        int i=10/0;
    }
}
public class L
{
    public static void main(String[] args)
    {
        M m1=new M();
        try
        {
            m1.test();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output: java.lang.ArithmeticException: / by
zero

Note: When a method of 1 class is used in another class without inheritance then we need to surround the method with try catch block.

Throws helps us to take up the memory address of the object and gives it to catch.

30. FILE HANDLING

Example:

```
import java.io.File;
public class A
{
    public static void main(String[] args)
    {
        File f1=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
        if(f1.mkdir())
        {
            System.out.println("File is created");
        }
        else
        {
            System.out.println("file not created");
        }
        if(f1.exists())
```

```
        {
            System.out.println("File exists");
        }
        else
        {
            System.out.println("file not exists");
        }
        if(f1.delete())
        {
            System.out.println("file deleted");
        }
        else
        {
            System.out.println(("file not deleted"));
        }
    }
}
```

Output: file not
created File exists file
deleted

Example:

```
import java.io.*;
public class B
{
    public static void main(String[] args)
    {
        File f1=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
        try
        {

            if(f1.createNewFile())
            {
                System.out.println("file is created");
            }
            else
            {
                System.out.println("File is not created");
            }
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Output: File is not
created

Example: public
class C
{
public static void main(String[] args)
{
File f1=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
try
{ if(f1.createNewFile())
{
System.out.println("File created");
}
else
{
System.out.println("File not created, bcz it exists");
}
}
catch(IOException e)
{
System.out.println(e.getMessage());
}
}
}

Output:
File not created, bcz it exists

Example:
import java.io.FileWriter; import
java.io.IOException; public class
D
{
public static void main(String[] args)
{ try {
FileWriter fw=new
FileWriter("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt"); fw.write("Hello
World, Hope you are fine"); fw.flush();
fw.close();
}
catch(IOException e)
{
System.out.println(e.getMessage());
}
}
}

Output:

Hello World, Hope you are fine

Example:

```
import java.io.*;
public class E
{
	public static void main(String[] args)
	{
		File f1=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
		try
		{
			FileReader fr=new FileReader(f1); char t[]
			= new char[(int)f1.length()];
			fr.read(t);
			String rv=new String(t);
			System.out.println(rv);
			System.out.println(rv.toUpperCase());
			fr.close();
		}
		catch(FileNotFoundException e)
		{
			System.out.println(e.getMessage());
		}
		catch(IOException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

Output:

Hello World, Hope you are fine HELLO
WORLD, HOPE YOU ARE FINE

Example:

```
import java.io.*;
public class F
{
	public static void main(String[] args)
	{
		File f=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
		try
		{
			FileWriter fw=new FileWriter(f);
			fw.write("ABCD"); fw.close();
```

```
                System.out.println("Writing Done!");

                FileReader fr=new FileReader(f); int a =
                fr.read(); //returns characters int no
                while(a!=-1) //End of file condition
                {
                    System.out.print((char)a); a=fr.read();
                } fr.close(); }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
}
```

Output:

Writing Done!0 ABCD

Example:

```
public
class G
{
    public static void main(String[] args)
    { try {
                FileWriter fw=new FileWriter("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
                fw.write("XYZ");
                fw.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
    }
}
```

Output:

Example:

```
import java.io.FileWriter; import
java.io.IOException; public class
H
{
    public static void main(String[] args)
    { try {
                FileWriter fw = new
                FileWriter("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt"); fw.write( );
                fw.close();
            }
            catch(IOException e)
```

```
        {
            e.printStackTrace();
        }
    }
}
```

Output:

The method write(int) in the type OutputStreamWriter is not applicable for the arguments ()

Example:

```
public
class I
{
    public static void main(String[] args)
    {
        try {
            FileWriter fw=new FileWriter("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
            fw.write("hello");
            fw.close();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

hello

Throws

Throws is a keyword which is used to re-throw the already existing generated exception to the other methods.

or

Throws is a keyword which is used to delegate the exception to other methods. 1.
Throw keyword should be used at the method invocation

2. Using throw keyword we can throw multiple exception.
3. To notify the uses of function or method about the run-time error the method generation for invalid value we can use throws keyword.

Example:

```
public
class J
{
    public static void main(String[] args)
    { try
        {
            int a[] = new int[999999999];
        }
        catch(OutOfMemoryError e)
        {
```

```
                System.out.println("caught");
            }
        }
}
```

Output: caught

What is the differences between exception class and error class of throw-able hierarchy?

Whenever there is a problem with external resources we get run-time error which are sub-class to error class.

Whenever there is a problem in the Java statement (some logical problems) there we get run-time error which are sub-class to exception class or run-time exception.

What is the difference between throw, throws and throw-able?

Throw is a keyword which is used to throw a new customized exception to the run-time environment.

If some business condition are not met and if Java cannot create any exceptions and throw it to the run-time environment by using throw keyword.

Throws is used to re-throw the already generated exception to other methods or caller methods.

Throw-able is a concrete class of java.lang package and throw-able class is the super-most class for all the exception related classes.

Throw-able catch block can handle any kind of exception.

Example:

```
import java.io.BufferedWriter;
import java.io.File; import
java.io.FileWriter;
public class K
{
    public static void main(String[] args) throws Exception
    {
        File f1=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
        FileWriter fw = new FileWriter(f1);
        BufferedWriter bw=new BufferedWriter(fw); String[]
        ref={"Dog","cat","sheep","cow"};
        for(int i=0;i<ref.length;i++)
        { bw.write(ref[i]);
            bw.newLine();
        }
        bw.close();
    }
        System.out.println("Writing done");
}
```

Output: Writing done

Example:

```
import java.io.BufferedReader;
import java.io.File; import
java.io.FileReader; public class L
{
    public static void main(String[] args) throws Exception
    {
        File f=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
        FileReader fr=new FileReader(f);
        BufferedReader br=new BufferedReader(fr);
        String s=br.readLine();
        {
            while(s!=null)
            {
                System.out.println(s);
                s=br.readLine();
            } br.close();
        }
    }
}
```

Output: Dog
cat sheep
Cow

Example:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File; import
java.io.FileReader; import
java.io.FileWriter; import
java.io.IOException; import
java.util.ArrayList;
import java.util.Collections;
public class M
{
    public static void main(String[] args) throws IOException
    {
        File f=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
        FileReader fr=new FileReader(f);
        BufferedReader br=new BufferedReader(fr);
        //ArrayList<String>list =new ArrayList<String>
        ArrayList<String> list = new ArrayList<String>();
        String s=br.readLine();
        while(s!=null)
        { list.add(s);
            s=br.readLine();
        }
```

```
            Collections.sort(list);
            FileWriter fw=new FileWriter(f);
            BufferedWriter bw=new BufferedWriter(fw);
            for(int i=0;i<list.size();i++)
            { bw.write(list.get(i));
                  bw.newLine();
            } br.close();
            br.close();
            System.out.println("file Updated");
      }
}
```

Output: file
Updated

Example: Program to remove duplicate name from File

```
import java.io.File; import java.io.FileInputStream; import
java.io.FileOutputStream; import java.io.IOException;
public class N
{
      public static void main(String[] args) throws IOException
      {
            File f=new File("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
            FileInputStream fin = new FileInputStream(f); byte[]
            barr = new byte[(int)f.length()];
            fin.read(barr); fin.close();
            FileOutputStream fout= new
      FileOutputStream("C:¥¥Users¥¥JYOTHISHANKAR¥¥Desktop¥¥Testing.txt");
            fout.write(barr);
            fout.close();.
            System.out.println("writing done");
      }
}
```

Output: writing done

Example:

```
import java.io.IOException; import
java.util.Scanner;
public class O
{
      public static void main(String[] args) throws IOException
      {
            String pwd="Qspiders";
            Scanner scan = new Scanner(System.in);
            System.out.println("Please Enter one number"); int a
            = scan.nextInt();
```

```
            System.out.println("Please Enter another number"); int
            b=scan.nextInt();
            System.out.println("Please enter password for result");
            String s=scan.nextLine();
            if(pwd.equals(s))
            {
                  int sum=a+b;
                  System.out.println("result is" +sum);
            }
            else
            {
                        System.out.println("sorry incorrect password");
            } scan.close();
      }
}
```

Output:

Example:

```
import java.io.IOException; import
java.util.Scanner;
public class P
{
      public static void main(String[] args) throws IOException
      {
            Scanner scan = new Scanner(System.in);
            System.out.println("Please enter one string");
            String s1=scan.nextLine();
            System.out.println("please enter one word"); String
            s2=scan.next();
            System.out.println("enter you replacement word");
            String s3=scan.next(); s1=s1.replaceAll(s1,
            s3); System.out.println(s1);
            scan.close();
      }
}
```

Output:

```
Please enter one string hi, hope
you doing Good please enter one
word Good enter you
replacement word bad bad
```

Example:

```
import java.util.ArrayList; import
java.util.Scanner; import java.util.*;
public class R
{
```

```
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in); System.out.println("Please
        Enter the size of Array");
        int size=scan.nextInt(); double[]
        darr = new double[7];
        System.out.println("please Enter " +size+ "elements");
        for(int i=0;i<size;i++)
        {
            darr[i]=scan.nextDouble();
        }
        System.out.println("The" +size+ "elements are");
        for(int i=0;i<size;i++)
        {
            System.out.println(darr[i]);
        }
        System.out.println("Sort?? Yes or No"); String
        opt=scan.next();
        if(opt.equalsIgnoreCase("yes"))
        {
            Arrays.sort(darr);
            for(int i=0;i<size;i++)
            {
                System.out.println(darr[i]);
            }
        }
        else
        {
            System.out.println("Thank you");
        } scan.close();
        }
}
```

Output:

```
Please Enter the size of Array
4
please Enter 4elements
1
3
5
7
The4elements are
1.0 3.0 5.0
7.0
Sort?? Yes or No yes
0.0 0.0 0.0
```

1.0

Example:

```
public class Bank
{
	public static void main(String[] args)
	{
		Scanner scan=new Scanner(System.in);
		Bank rv=new Bank();
		int opt; do
		{
			System.out.println("1: balance¥t 2: Deposit");
			System.out.println("3:WithDraw¥t 4: Exit"); System.out.println("What's
			your option?");
			opt=scan.nextInt();
			switch(opt)
			{
			case 1 : rv.balance(); System.out.println();
					rv.balance();
				break;
			case 2: System.out.println("Please Enter deposit amount");
					int damount=scan.nextInt(); rv.deposit(damount);
				break;
			case 3: System.out.println("please enter the withdraw amount");
					int wamount =scan.nextInt(); rv.withdraw(wamount);
				break;
			case 4: System.out.println("Thank You"); break;
			default:System.out.println("invalid option");
			}
		}
		while(opt!=4);
		{ scan.close();
		}
	}
	private void withdraw(int wamount)
	{

	}
	private void deposit(int damount)
	{
	}
	private void balance()
	{

	}
}
```

Output:

1: balance 2: Deposit
3:WithDraw 4: Exit What's
your option?

Example: public

```
class S
{
    public static void main(String[] args)
    {
        Scanner scan=new Scanner(System.in);
        System.out.println("Please Enter one number"); int
        a= scan.nextInt();
        System.out.println("please enter another number");
        int b=scan.nextInt(); if(a>=b)
        {
            System.out.println("The result is " +(a-b));
        }
        else
        {
            throw new ArithmeticException("first No. should be greater than second no.");
        }
    }
}
```

Output:

Please Enter one number
5
please enter another number
5
The result is 0

31. JAVA MEMORY MANAGEMENT - Garbage Collection:

Garbage collection is a process of removing the abandoned objects from the heap memory.
Abandoned object means the object without any references.
The abandoned object will be removed from the heap memory by using finalize() of that abandoned object by garbage collection.

Garbage Collector is a thread which will be always running in the background.
Garbage Collector is a daemon thread, which will execute whenever the system is free.

Daemon thread means lowest priority thread.
We can predict when exactly an object is eligible for garbage collection.
We cannot predict when exactly garbage collections happens.
While developing the program if you wanted to perform garbage collection then we can use System.gc();

Example:

```
class Z
{
```

```
    @override
    protected void finalize() throws Throwable
    {
        System.out.println("running finding method"); super.finalize();
    }
}
public class T
{
    public static void main(String[] args)
    {
        Z z1=new Z(); z1=new
        Z();
        System.gc();
    }
}
```

Output:

running finding method

Example: class D

```
{       int i;    } class C
{
    void test1()
    {
        System.out.println("non-static method");
    }
}
public class B
{
    public static void main(String[] args)
    {
        System.out.println(new D().i);
        new C().test1();
        System.out.println(new String("qspiders").toUpperCase());
        System.out.println("Jspiders".length());
    }
}
```

Output: 0 non-static method

QSPIDERS 8

Example: class Sample

```
{
    static String test()
    {
        return "java Programming";
```

```
    }
}
class Demo
{
    public String test2()
    {
        return "selenium";
    }
}
public class Z
{
    public static void main(String[] args)
    {
        System.out.println(Sample.test().length());
          System.out.println(new Demo().test2().length());
    }
}
```

Output:

16

8

32. FINAL, FINALLY, FINALIZE

Final:

1. Is a keyword.
2. If you want to declare constant variables where in the value cannot change, then those variable should be declared with the keyword “final”.
3. Final variable value cannot be changed or overridden.
4. Both local and global variable can be final.
5. Global final variable should be initialized at the time of initialization itself.
6. Local final variable can be declared once and initialized later.
7. In terms of inheritance, final keyword us used to avoid overriding.
8. Final class cannot be inherited.
9. Final class methods cannot be overridden because for method overriding inheritance is mandatory.

Example:

```
public
class A
{
    public static void main(String[] args)
    {
        final int i=10; i=20;
        System.out.println(i);
    }
}
```

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The final local variable i cannot be assigned. It must be blank and not using a compound assignment

Note: If you make a variable has final then we can never re-initialize that variable.

Example: public
class B
{
public static void main(String[] args)
{
final int i=10; i=10;//error
System.out.println(i);
}
}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The final local variable i cannot be assigned. It must be blank and not using a compound assignment

Note:

Static and non-static variable if made Final then ts initialization is mandatory or else we will get blank field error.

Example:

{
final int i;//error because not initialized
public static void main(String[] args)
{
final int j;//local variables should be initialized before usage System.out.println(j);
}
}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The local variable j may not have been initialized

Note:

In case of local variable make it final there is no error, but using it without initialization will throw error.

Example: public
class D
{ final static int i;
public static void main(String[] args)
{ final int j;
System.out.println(j);
}
}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The local variable j may not have been initialized

Note: In final, copying value of variable is possible. Final keyword make the size fixed i.e., cannot be altered.

Example:
```
public
class E
{
    public void test(final int i)
    {
        i=30;//The final local variable i cannot be assigned. It must be blank and not using a compound assignment
        System.out.println(i);
    }
    public static void main(String[] args)
    {
        E e1=new E(); e1.test(10);
    }
}
```
Output:
The final local variable i cannot be assigned. It must be blank and not using a compound assignment

Note: If you make an array as final then its size cannot be altered.

Example:
```
public
class F
{
    public static void main(String[] args)
    {
        final int[] a=new int[5];
        a[0]=10; a[0]=20;
        System.out.println(a[0]); //20
    }
}
```
Output: 20

Example:
```
public
class G
{
    public static void main(String[] args)
    {
        final int[] a=new int[5];
        a=new int[2];
    }
}
```
Output:
The final local variable a cannot be assigned. It must be blank and not using a compound assignment

Note:

1. When array is final, its size cannot be changed.
2. When array is not final, then size can be altered.
3. When (String[] args) is final, then its size cannot be altered.
4. args[] is a take string no number allowed

Example: public

```
class H
{
	public static void main(String[] args)
	{
		args=new String[3]; args[0]="10";
		System.out.println(args[0]);
	}
}
```

Output: 10

Note:

1. Inheritance of final class is not possible.
2. Final Variable value cannot be altered.
3. Final Array size cannot be altered, but value can be altered.
4. Final Class, inheritance is not possible and value cannot be inherited.

Example: final class

```
J
{
	int i=10;
}

final class I extends J// the type I cannot sub class to final
{
	public static void main(String[] args)
	{
		J j1=new J();
		System.out.println(j1.i);
	}
}
```

Output:

Unresolved compilation problem the type
I cannot sub class to final

Note: A final class can inherit the members of non-final class.

Example:

```
class L
{
	int i=20;
}
final class K extends L
{
```

```
    public static void main(String[] args)
    {
        L l1=new L();
        System.out.println(l1.i);
    }
}
```

Output: 20

Note: Whether local,static,non-static variable, value cannot be altered.

Example:

```
public class Y extends Z
{
    public static void main(String[] args)
    {
        Y y1=new Y();
        y1.i=200;
    }
}
```

Output:

ion in thread "main" java.lang.Error: Unresolved compilation problem: The
final field Z.i cannot be assigned

Note:

1. For overriding access specifier can be different, but return type should be same.
2. When you make a method final and not use void then you get Error.

Example:

```
public class W
extends X
{
    public int test()//not a void method when declared test(), so error.
    {
        System.out.println(""); return 30;
    }
    public static void main(String[] args)
    {
        W w1=new W();
        w1.test();
    }
}
```

Output:

Cannot override the final method from X

Finally:

Finally is a block which will be used in Exception Handling.

Finally block will be executed irrespective of Exception.

To execute certain mandatory statement without fail we will use Finally block.

Closing the connection established between database and file system of the computer will be done in finally block, so in any scenario connection will be closed.

Finalize():

Finalize is a non-static method in object class which will be used for garbage collection by garbage collections thread.

Whenever we use System.exit() statement, then finally block will not execute.

33. MULTI THREADING

Multi-tasking: Performing multiple task or operation concurrently or simultaneously is called Multi-tasking. Multi_tasking can be achieved in 2 ways,

1. Multi-Processing
2. Multi-Threading

1. Multi-Processing:

   Executing multiple processes simultaneously is called as Multi-processing.
   Executing multiple application simultaneously is called Multi-processing. Here 3-4 milli second of processor time will be shared between multiple application.

2. Multi-Threading:

   Dividing an application into multiple parts and allocating the separate processor time and memory for each part of the same application is called as Multi-Threading.

THREAD

Thread is a execution instance which will have its own processor time and memory.
Whenever a core java program is executed 3 threads will be created automatically, they are:

1. Main Thread.
2. Thread Scheduler.
3. Garbage Collection.

1. Main Thread:
   - ✓ Main thread responsibility is to execute main method of program.
   - ✓ Main method is different and main thread is different. ・ Main method is foreground process.
2. Thread Scheduler:
   - ✓ Thread Scheduler responsibility is to perform Thread registration.
   - ✓ Thread registration means allocating the separate processor time and memory for the thread.
3. Garbage Collector:
   - ✓ GC responsibility is to perform Garbage collection and it is a lowest priority background thread.
   - ✓ In java, there is a built-in class called Thread, it is a sub class to Object class.

Example:

```
package Multithreading;
public class Tester1
{
    public static void main(String[] args)
    {
```

```
        System.out.println("Main Starts"); try
        {
            Thread.sleep(20000);
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("Main Ends");
    }
}
```

Output:

```
Main Starts
Main Ends
```

Example:

```
¥public class Tester2
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main Starts"); for(int
        i=1;i<=5;i++)
        {
            System.out.println(i);
            Thread.sleep(1000);
        }
        System.out.println("Main Ends");
    }
}
```

Output:

```
Main Starts
1
2
3
4
5
Main Ends
```

Example:

```
public class Tester3
{
    public static void main(String[] args)
    {
        Thread t1=Thread.currentThread();
        System.out.println("Name:" +t1.getName());
```

```
            System.out.println("Id:" +t1.getId()); System.out.println("Priority:"
            +t1.getPriority()); t1.setName("initiator");
            t1.setPriority(10);
            System.out.println("Name" +t1.getName());
            System.out.println("Id:" +t1.getId());
            System.out.println("Priority:" +t1.getPriority());
      }
}
```

Output:

```
Name:main
Id:1
Priority:5
Nameinitiator
Id:1
Priority:10
```

Example:

```
public class Tester4
{
      public static void main(String[] args)
      {
            Thread t1=Thread.currentThread();
            t1.setName("Initiator");
            for(int i=1;i<4;i++)
            {
                  System.out.println(t1.getName()+"::" +i); try
                  {
                        Thread.sleep(1000);
                  }
                  catch(InterruptedException e)
                  {
                        e.printStackTrace();
                  }
             }
      }
}
```

Output:

```
Initiator::1
Initiator::2
Initiator::3
```

Note:

1. Thread is a concrete class of java.lang package.
2. Thread class have a lot of static and non-static method through which we can work with Thread.
3. Thread class is a sub class to object class.

4. Thread class have 2 important static method,
    a) Sleep():
    b) currentThread():

a)sleep():

- ✓ sleep is a static method of thread class which will receive millisec as input in long form.
- ✓ Whenever sleep() is called on any thread, that particular thread will stop its execution for supplied millisec.
- ✓ Usage: Thread.sleep(1000); Here, 1000 millisec means 1 sec.
- ✓ sleep() will throw a checked exception, so whenever we are using thread.sleep() we should handle it at compile time itself either through try-catch or throws keyword.

b)**currentThread():**

- ✓ It is a static method of thread class.
- ✓ Whenever currentThread class is used on any thread it will give the object of thread on which it is called.
- ✓ Using the object returned by currentThread we can explore the property o any Thread. ・ Usage: Thread t1=Thread.currentThread();

Any Thread will have 3 properties:

1. Name
2. Id
3. Priority

1. Name:
    - ✓ Any thread will have name.
    - ✓ Main threads name is Main itself.
    - ✓ For user threads, the name will be thread(0), thread(1), thread(2) etc., ・ Any threads name can be changed.
2. Id:
    - ✓ Id is a unique long number which cannot be duplicated.
    - ✓ Id will be used to identify a thread uniquely among the group of threads.
3. Priority:
    - ✓ Any thread will have priority between 1 to 10.
    - ✓ The default priority of all thread will be 5.
    - ✓ The priority of thread can be changed.

Example:

```
package Multithreading;
class A extends Thread
{
@Override
public void run()
{
    Thread t1=Thread.currentThread();
    for(int i=1;i<=5;i++)
```

```
        {
            System.out.println(t1.getName()+ "::" +i); try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }
}
}
class B extends Thread
{
        @Override
        public void run()
        {
            Thread t1=Thread.currentThread();
            for(int i=1;i<=5;i++)
            {
                System.out.println(t1.getName()+ "::" +i); try
                {
                    Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        }
}
public class Tester5
{
        public static void main(String[] args)
        {
            A       a1=new A();
            a1.setName(" A Thread");
            a1.start();
            B       b1=new B();
            b1.setName("B thread");
            b1.start();
            Thread t1=Thread.currentThread(); t1.setName("Initiator");
            for(int i=1;i<=5;i++)
            {
                System.out.println(t1.getName()+ "::" +i);
                try
```

```
                {
                        Thread.sleep(1000);
                }
                   catch(InterruptedException e)
                {
                        e.printStackTrace();
                }
            }
        }
}
```

Output:

```
A Thread::1
Initiator::1
B thread::1
B thread::2 A
 Thread::2
Initiator::2
Initiator::3 A
 Thread::3
B thread::3
Initiator::4 A
 Thread::4
B thread::4
Initiator::5
B thread::5
 A Thread::5
```

Example:

```
public class Tester6
{
      public static void main(String[] args)
      {
            EvenNumber a1=new EvenNumber();
            a1.setName("even no"); a1.start();
            OddNumber b1=new OddNumber();
            b1.setName("Odd number");
            b1.start();
      }
}
class EvenNumber extends Thread
{
      @Override public void
      run()
      {
            Thread t1=Thread.currentThread();
```

```
            for(int i=1;i<=20;i++)
            {
                  System.out.println(t1.getName()+ "::" +i);
            }
      }
}
class OddNumber extends Thread
{
      @Override public void
      run()
      {
            Thread t1=Thread.currentThread();
            for(int i=1;i<=30;i++)
            {
                  System.out.println(t1.getName()+ "::" +i);
            }

      }
}
```

Output: even no::1
even no::2 even
no::3 even no::4
even no::5 even
no::6 even no::7
even no::8 even
no::9 even no::10
Odd number::1
Odd number::2
Odd number::3
Odd number::4
Odd number::5
Odd number::6
Odd number::7
Odd number::8
Odd number::9
Odd number::10
Odd number::11
Odd number::12
Odd number::13
Odd number::14
Odd number::15
Odd number::16
Odd number::17
Odd number::18
Odd number::19
Odd number::20

----------**Note:**

1. Thread class is having 2 important non-static methods. They are: start(); run();
2. Whenever any class extends Thread class then that class will inherit both start and run method.
3. We should override the inherited run method.
4. The task which has to be executed by the thread should be developed in the run method while overriding.
5. We should always call the start method to initiate the thread.
6. Start() will perform 2 operations.

a)Operation 1: It will perform Thread registration through Thread Scheduler.
b)Operation 2: It will call the overridden run method.

Thread Synchronization

Whenever multiple threads will execute the same method or resources then data corruption happen, so we should perform Thread Synchronization.

To perform thread synchronization, the method should be declared with the keyword Synchronize.

Synchronize method cannot be executed by multiple threads simultaneously. Synchronized methods are also called as Thread Safe methods.

Example: If 2 threads are using the same transfer function and function is not synchronized then data corruption might happen on balance data.
Example: Vector class of collection is called Thread safe because all the method of vector class is Synchronized. i.e., if any thread is using add() on vector object another thread cannot remove the data simultaneously.

Note: All the synchronized method in eclipse will have a lock method.

Note:
1.Every object in java will have one object lock and only one lock per object.
2.Object lock is related to threads.
3.If nay threads need to execute synchronized methods then it should have object lock.
4.After completely executing synchronized method, the thread will release the object lock. So synchronized method cannot be executed by multiple threads together.

Example:

```
A.class package
ThreadSync;
public class A
{
    synchronized void test1()
    {
        Thread t1=new Thread().currentThread();
        System.out.println(t1.getName() +"Started test1");
        for(int i=1;i<=5;i++)
        {
            System.out.println(t1.getName() +"::" +i); try
```

```
                {
                    Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
                  System.out.println(t1.getName()+ "completed test1");
            }
        }
}
```

B.class package

```
ThreadSync;
public class B extends Thread
{
    A a1;
    B(A a1)
    {
        this.a1=a1;
    }
    @Override public void
    run()
    { a1.test1();
    }
}
```

C.class package

```
ThreadSync;
public class C extends Thread
{
    A a1;
    C(A a1)
    {
        this.a1=a1;
    }
    @Override public void
    run()
    { a1.test1();
    }
}
```

Tester.class package

```
ThreadSync;
public class Tester4
{
    public static void main(String[] args)
    {
        A a1=new A();
```

```
        B b1=new B(a1); C c1=new     C(a1);
        b1.setName("B Thread");
        c1.setName("C Thread");
        b1.start(); c1.start();
    }
}
```

Output:

```
B ThreadStarted test1
B Thread::1
B Threadcompleted test1
B Thread::2
B Threadcompleted test1
B Thread::3
B Threadcompleted test1
B Thread::4
B Threadcompleted test1
B Thread::5
B Threadcompleted test1
C ThreadStarted test1
C Thread::1
C Threadcompleted test1
C Thread::2
C Threadcompleted test1
C Thread::3
C Threadcompleted test1
C Thread::4
C Threadcompleted test1
C Thread::5
C Threadcompleted test1
```

34. GENERICS

Generics was introduced in Java 1.5.
Generics can create a variable in a way that can store any kind of data.

Example: Non-Generic class class A

```
{ int i;
    public static void main(String[] args)
    {
        A a1=new A();
        a1.i=10;
        System.out.println(a1.i);
    }
}
```

Output:

10

Example: Generic class

```
public class B<x>
{ x i;
    public static void main(String[] args)
    {
        B b1=new B();
        b1.i=10.5;
        System.out.println(b1.i);
    }
}
```

Output:

10.5

Note:

Generic changes according to the situation.
X can take any alphabet.
Instead of X, any alphabet can be used.

Definition:

A Generic class helps us to create a variable such that any kind of value can be stored in the variable, because the data type of variable is decided based on the kind of value is stored in the variable.

Example: public class

```
    C<Z>
    { Z i;
        @SuppressWarnings("rawtypes")
        public static void main(String[] args)
        {
```

```
            C c1=new C();
            c1.i=10.3f; c1.i=20;
            c1.i=30;
            System.out.println(c1.i);
        }
    }
```

Output: 30

Note: The latest value of I variable will be printed.

Example: Multiple Generics(any number of Generic representations)

```
public class D
<X,Y,Z>
{
    X a; Y b; Z
    c;
    public static void main(String[] args)
    {
        D d1=new D();
        {
            d1.a=10; d1.b=20.3;
            d1.c=30.3f;
            System.out.println(d1.a);
            System.out.println(d1.b);
            System.out.println(d1.c);
        }
    }
}
```

Output:

10
20.3
30.3

Example:

```
public class F<X>
{
    X i;
    X j; X k;
    public static void main(String[] args)
    {
        F f1=new F(); f1.i=30;
        f1.j=40.4; f1.k=50.5f;
        System.out.println(f1.i);
        System.out.println(f1.j);
        System.out.println(f1.k);
    }
}
```

Output:

30
40.4
50.5

Note:

If a method is made a Generic then it will get potential to retuern any kind of value.
When a normal data type is replaced by generic data type, then it can retuen value.

Example:

```
public class E<X>
{
    X i;//generic cannot be applied on local variable
    E e1; public X test()
    {
        e1=new E(); e1.i=60.6f;
        return i;
    }
    public static void main(String[] args)
    {
        E e2=new E();
        e2.test();
        System.out.println("" +e2.i);
        System.out.println("" +e2.test());
    }
}
```

Output: null
null

Note:

1) Generic cannot be applied on local variable.
2) Generic cannot be applied on static variables but can be applied to non-static variable

Example:

```
public class G<X>
{
    X i;
    X j; X k;
    public static X test() //Cannot be static
    {
        g1=new G();
        g1.i=90.9f;
        return i;//cannot make reference to the non-static field i
    }
    public static void main(String[] args)
    {
        G g2=new G();
```

```
        g2.test();
        System.out.println(g1.i);
    }
}
```

Output:

35. INNER CLASS

A class created within another class is a Inner class. A class within other class is a Inner class.

3 types of Inner class:

1. Local Inner Class.
2. Static Inner Class.
3. Anonymous Inner class.

1. Local Inner class:

The class within the outer class which is called as Inner class.

To access members of local Inner class:

1) First create the object of outer class. A a1=new A();
2) Take the reference of outer class and then create object of inner class and then access member of inner class.

   B b1=a1.new B();

Example:

```
public
class A
{
    class B
    {
        int i=0;
    }
    public static void main(String[] args)
    {
        B b1=new B();
        System.out.println(b1.i);
    }
}
```

Output:

No enclosing instance of type A is accessible. Must qualify the allocation with an enclosing instance of type A (e.g. x.new A() where x is an instance of A).

Example:

```
public
class B
{
    class C
    {
```

```
        int i=10;
    }
    public static void main(String[] args)
    {
        B b1=new B();
        C c1=b1.new C();
        System.out.println(c1.i);
    }
}
```

Output:

10

Example: public

```
class C
{
    class D
    {
        public void test()
        {
            System.out.println("from test");
        }
    }
    public static void main(String[] args)
    {
        C c1=new C(); D
        d1=c1.new D();
        d1.test();
    }
}
```

Output: from test

Note:

1.We cannot create static local member in an Inner class.

2. Inner class should only have non-static method.

Example: public

```
class D
{
    class E
    {
        static int i=10;// The field i cannot be declared static in a non-static inner type, unleass
initialized with a constant expression
        public static void test()
        {
            System.out.println("From test()");
        }
```

```
    }
    public static void main(String[] args)
    {
        D d1=new D();
        E e1=d1.new E();
        System.out.println(e1.i);//The static field D.E.i should be accessed in a Static way
    }
}
```

Output:

The field i cannot be declared static in a non-static inner type, unless initialized with a constant expression

The method test cannot be declared static; static methods can only be declared in a static or top level type

Example: public

```
class E
{
    int j=20; class F
    {
        int i=10;
    }
    public static void main(String[] args)
    {
        E e1=new E();
        F f1=e1.new F();
        System.out.println(f1.i);
        System.out.println(f1.j);// By creating an object of local inner class we cannot access the
members of outer class
    }
}
```

Output:

j cannot be resolved or is not a field

Note:

By creating an object of local inner class we cannot access the members of outer class

Example: public class F

```
{
    class G
    { G()
        {
            System.out.println("from G");
        }
    }
    public static void main(String[] args)
    {
        F f1=new F();
```

```
            f1.new G(); // or G g1=f1.new G();

        }
}
```

Output: from G

Note:

We can create a constructor inside a local inner class.

Example:

```
public
class G
{
    static int i=10;
    class H extends G
    {
        static int j=20; static int
        i=20;
    }
    public static void main(String[] args)
    {
        G g1=new G();
        H h1=g1.new H();
        System.out.println(h1.i);
    }
}
```

Output:

The field j cannot be declared static in a non-static inner type, unless initialized with a constant expression

The field i cannot be declared static in a non-static inner type, unless initialized with a constant expression

Note:

Static members of outer class can be inherited into local inner class but then static members cannot be created in local inner class.

Example:

```
public
class H
{
    class I
    {
        int i=10;
    }
    class J
    {
        int j=20;
    }
    public static void main(String[] args)
    {
```

```
            H h1=new H();
            I  i1=h1.new I();
            System.out.println(i1.i); J
            j1=h1.new J();
            System.out.println(j1.j);
      }
}
```

Output:

10
20

Note:

We can create more than 1 local inner class inside a same class.
When we compile the inner class program we will get class as follows,
A$B.class A$C.class
A.class
But the above program will be saved as a single java file.

Rule 1:

Inheritance between two inner class is possible

Example: public class I

```
{
      class J
      {
            int i=20;
      }
      class K extends J
      {
            int j=40;
      }
      public static void main(String[] args)
      {
            I i1=ne K();
            System.outw I();
            K k1=i1.new.println(k1.i);
            System.out.println(k1.j);
      }
}
```

Output:

20
40

Example: public

```
class J
{
      class K
      {
```

```
        {
            System.out.println("from IIB"); //Called first
        }
        K()
        {
            System.out.println("constructor");
        }
    }
    public static void main(String[] args)
    {
        J j1=new J();
        j1.new K();//K k1=j1.new K();
    }
}
```

Output: from IIB
constructor

Note:
When IIB and constructor are both together then upon object creation IIB will be called first and then constructor.

Note:
When a instance variable is created make sure declaration is outside if you create a variable inside IIB then it becomes local variable.
Anything created within braces are local, but outside braces and within class is instance variables.

Example:

```
public class K
{ class L
    { int i;
        { int j=20;
            System.out.println("from IIB");
        }
        L()
        {
            System.out.println("from constructor");
        }
    }
    public static void main(String[] args)
    {
        K  k1=new K();
        L  l1=k1.new L();
        System.out.println(l1.i);
    }
}
```

Output:

from IIB from constructor
0

Note:
IIB can be created inside local inner class

Rule 2:
Incomplete methods in interface should be completed in class by inheriting and overriding.

Example:

```
public interface AA
{
      public void test();
}
public class L
{
      class M implements AA
      {
            public void test()
            {
                  System.out.println("from test()");
            }
      }
      public static void main(String[] args)
      {
            L l1=new L();
            M      m1=l1.new M(); m1.test();
      }
}
```

Output: from
test()
Class M can extend member of class L.
Class M can implement member of interface AA.

1. Only IIB can be created in Inner class.

2. Static Inner Class. Syntax: public class A
```
{
    static class B
    { static int i; int j=20;
    }
}
```

Note:

1. In comparison to Local Inner class, in static Inner class both static and non-static can be created.
2. To access the members(because they are static) you can directly create the object of the static class.

Ex: System.out.println("B.i"); System.out.println(b1.j);

3. In this class we can have both static and non-static members.

Example:
```
public class A
{
    static class B
    { static int i; int j=20;
    }
    public static void main(String[] args)
    {
        B b1=new B();
        System.out.println(B.i);//SIB
        System.out.println(b1.j);//IIB
    }
}
```
Output:

0

20

Note:

Inside static class we can create SIB, IIB, constructor.

Example:
```
public class C
{
    static class D
    { static
        {
            System.out.println("SIB");
        }
        D()
        {
            System.out.println("constructor");
        }
```

```
        {
            System.out.println("IIB");
        }
        public static void main(String[] args)
        {
            D d1=new D();
        }
    }
}
```

Output:

SIB

IIB Constructor

Note:

Inheriting member of outer class to static Inner class.

Example: public class E

```
{
    int i=10;
    static class F extends E
    {

    }
    public static void main(String[] args)
    {
        F f1=new F();
        System.out.println(f1.i);
    }
}
```

Output:

10

Note: We cannot create object of static Inner class and access the members of outer class without inheritance.

Example: public class G

```
{
    int i=100;
    static class H //extends G
    {

    }
    public static void main(String[] args)
    {
        H h1=new H();
        System.out.println(h1.i);
    }
}
```

Output:

i cannot be resolved or is not a field

_____ We can have more than 1 static inner class

Example: public class I

```
{
    static class J
    {
        int j=10;
    }
    static class K
    {
        int k=20;
    }
    public static void main(String[] args)
    {
        J j1=new J();
        K k1=new K();
        System.out.println(j1.j);
        System.out.println(k1.k);
    }
}
```

Output: 10

20

Note: Inheritance between two static Inner class is possible

Example: public class L

```
{
    static class M
    {
        int i=80;
    }
    static class N extends M
    {
        int j=90;
    }
    public static void main(String[] args)
    {
        N n1=new N();
        System.out.println(n1.i);
        System.out.println(n1.j);
    }
}
```

Output: 80

90

_____ Creating the object of Static Inner class cannot access the members of Outer class.

Example: public class O

```
{
    int i=10; static class P
    {
        int j=20;
    }
    public static void main(String[] args)
    {
        P p1=new P();
        System.out.println(p1.i);//Object of inner class cannot access the member of Outer class
    }
}
```

Output: i cannot be resolved or is not a field

Note: Static Inner classes can implement Interfaces

Example: public class Q

```
{
    static class R implements QQ
    {
        public void test()
        {
            System.out.println("from Interface test()");
        }
    }
    public static void main(String[] args) {
        R r1=new R();
        r1.test();
    }
}
```

Output: from Interface test()

Note: Inheriting local Inner class members into static Inner class is not possible.

Example: public class S

```
{
    class T
    {
        int i=100;
    }
    static class U extends T
    {

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

```
        U u1=new U();
        System.out.println(u1.i);
    }
}
```

Output:

No enclosing instance of type S is available due to some intermediate constructor invocation.

_____ Inheriting static Inner class members into Local Inner class is possible.

Example:

```
public class V
{
    static class W
    {
        static int j=60;
                int i=70;
    }
    class X extends W
    {

    }
    public static void main(String[] args) {
        V v1=new V();
        X x1=v1.new X();
        System.out.println(x1.i);
        System.out.println(X.j);//X will be converted to W, because conversion takes place
        }
}
```

Output: 70

60

Differences:

Static Inner Class	Local Inner Class
1. Can consist all the members	1. Can consists of only non-static members
2. Directly create the Object then access the members.	2. Create Object of Outer class, take its reference then access Inner class.

Rule of Inner class:

1. We cannot create a main method inside a local Inner class as main method is static.
2. We can create main method inside a static Inner class but then it will execute when we call it.

Example:

```
public class Y
{
    static class Z
    {
        int j=10;
        public static void main(String[] args)
        {
            int i=100;
            System.out.println(i);
```

```
        }
    }
    public static void main(String[] args)
    {
        Z z1=new Z();
        System.out.println(z1.j);//calling statement
        Z.main(null);
    }
}
```

Output:

100

3). Anonymous Class:

A class without any name is called as Anonymous class.

Anonymous class should immediately precede with braces and semi-colon.

Without creating object anonymous class cannot be created.

Syntax: class A

```
{
    public static void main(String[] args)
    {
        A a1=new A();
        {

        };
```

Anonymous class

Note:

1. When the class has name use implement.
2. When the class dies not have name, the syntax of implements will be **new B(){ };** (this class implements interface) **Example:**

```
public class A
{
    public static void main(String[] args)
    {
        AA b1=new AA(){
            public void test()
            {
              System.out.println("from B");
            }
        };
        b1.test();
    }
}
```

Output: from B

Example: public class B

```
{
    public void test1()
    {
```

```
            System.out.println("from B");
      }
}
public class C
{
      public static void main(String[] args)
      {
            B b1=new B(){//Anonymous class extends B public void
                              test()
                              {
                                    System.out.println("from B");
                              }
                        };
                        b1.test1();
      }
}
```

Output: from B

———

1. In class, inherit complete method and overriding complete method.
2. In interface, inherit incomplete method and override with complete method **Note:**

1. Anonymous class automatically inherits the members of the object before it.
2. new()(member of the object will be created and inherited).
3. Anonymous class actually works on the principle of Inheritance.

Example:

```
class D
{
      public void test()
      {
            System.out.println("from test()");
      }
}
public class E
{
      public static void main(String[] args)
      {
            D d1=new D(){
                        };
                        d1.test();
      }
}
```

Output:

from test()

Note: Anonymous class is capable of inheriting members and then only make it accessible.

Create local Inner class to use with Anonymous class

1. Create Object, B | b1;
2. New B(){ | };To access Member
3. First create the object of first class(i.e., A a1=new A();) Anonymous class can inherit the members of local inner class as well.

Example: public class F

```
{
    class G
    {
        int i=10;
    }
    public static void main(String[] args)
    {
        F f1=new F();//creating object of F class
        G g1=f1.new G(){

                    };
                    System.out.println(g1.i);
        }
}
```

Output:

10

Note:

1. Anonymous class can inherit members of any other class.
2. Anonymous class can inherit interfaces.
3. Anonymous class can inherit local inner class.
4. Anonymous class can inherit static Inner class.
5. But cannot inherit members of Outer class.

Example: public class H

```
{
    static class I
    {
        public void test()
        {
            System.out.println("test-B");
        }
    }
    public static void main(String[] args)
    {
        I i1=new I(){
                        public void test()
                    {
                            System.out.println("test-anonymous");
                    } }; i1.test();
    }
```

```
}
```

Output: test-anonymous

36. INTERVIEW PROGRAMS

Example: Program to find ArmStrong number

```
public class ArmStrong
{
    static int sumofDigitsCube(int num)
    {
        int res=0;//173
        while(num!=0)//371
        {
            int temp=num%10;//371%10=71; res=res+(temp*temp*temp);
            num=num/10;
        }
        return res;
    }
    static void checkArmStrong(int num,int res)
    {
        if(num==res)
        {
            System.out.println("the number " +num+ " is ArmStrong");
        }
        else
        {
            System.out.println("the number " +num+ "is not ArmStrong");
        }
    }
    public static void main(String[] args) { int num=371; int
        res=ArmStrong.sumofDigitsCube(num);
        ArmStrong.checkArmStrong(num, res);
    }
}
```

Output: the number 371 is ArmStrong.

Example: Program to reverse number

```
public class Reverse
{
    static int reverse(int num)
    {
        int res=0; while(num!=0)
        {
            int temp=num*10;
            res=temp+(res*10);
            num=num/10;
        } return res;
    }
    public static void main(String[] args)
```

```
    {
        int num=354;
        int res=Reverse.reverse(354); System.out.println("Number is " +num);
        System.out.println("Reverse number is " +res);
    }
}
```

Output:

Example: Program to reverse a number

```
public class Palindrome
{
    static int reverse(int num)
    {
        int res=0;
        while(num!=0)
        {
            int temp=num%10; res=temp+(res*10);
            num=num/10;
        }
        return res;
    }
    static void checkPalindrome(int num,int res)
    {
        if(num==res)
        {
            System.out.println("the number is " +num+ " is palindrome");
        }
        else
        {
                                    System.out.println("The number is " +num+ " is not palindrome");
        }
    }
    public static void main(String[] args)
    {
        int num=121;
        int res=Palindrome.reverse(num); Palindrome.checkPalindrome(num, res);
    }
}
```

Output:

the number is 121 is palindrome

Example: class Adam

```
{
    static int reverse(int num)
    {
        int res=0;
        while(num!=0)
        {
            int temp=num%10; res=temp+(res*10);
            num=num/10;
        }
        return res;
    }
    static int square(int num)
    {
        int res;
        res=num*num;
        return res;
    }
    public static void checkAdam(int num,int res)
    {
        if(num==res)
        {
            System.out.println("the number is Adam");
        }
        else
        {
                                    System.out.println("The number is not Adam");
        }
    }
    public static void main(String[] args)
    {
        int num=12;
        System.out.println("the no is " +num); int
        num1=Adam.square(num); int res1=Adam.reverse(num); int
        num2=Adam.square((res1)); int res2=Adam.reverse(num2);
        Adam.checkAdam(num1, num2);
    }
}
```

Output: the no is 12 The number is not Adam

the no is 121 the number is Adam