

JAVA FULL STACK DEVELOPMENT

DETAILED HANDWRITTEN NOTES

PART I – CORE JAVA

Written by – Vivekanand Vernekar - [Linkedin](#)

For Programs and other Resources - [Github](#)

TOPICS COVERED:

- Programming Fundamentals
- Object Oriented Principles
- Exception Handling
- Multi threading
- File Handling

JAVA

First code in Java:

Class FirstCode

```
{
    for visibility           to receive command line arguments,
    public static void main (String a[])
    {
        System.out.print ("Hello World");
    }
}
```

Two things to do to run java program.

1. javac <file name>
2. java <class name>

- * When you compile the code internally it will create a file which is ".class" file.
- * When you make the class as public then you need to name the Classname same as file name.
- * In one file you can create multiple classes, and when you compile it multiple ".class" files are created i.e. for every class a .class file is created.

.jar - Java Archive : It's a file format based on the popular ZIP file format and is used for aggregating many files into one.

Main Method in Java :

In Java the main() method is the starting point from where compiler starts program execution.

Four things the method will have:

1. name
2. parameter
3. body
4. return type.

Difference between war file and .class file?

- Class file: Source code that is converted by a compiler that is a .class file
- war file: Collection of many .class files + .html files which is compressed we call as .war file
- Jar file: Collection of many .class files which is compressed we call as .jar file.

OOPS (Basic Introduction)

- * It Stands for Object Orientation Principles
- * Object: real time instance like Car, Student, Employee etc.
- * Every Object in real time will have two parts:
 - (a) what it does
 - (b) what it has

Java Code:

Class Car

{

// Has part of an object is represented as a "variable".

String brandName;

int noOfWheels;

// Does - part of an object is represented through "methods"

public void move()

{ // logic of moving a vehicle

{

public void accelerate()

{ // logic of acceleration

{

{

Identifier: * it is a name in java program

* it can be a class name, method name, variable name and label name.

Ex 1.

Class Test

```
{
    public static void main(String [args])
        { int a = 10; }
}
```

Total 5 identifiers.

Ex 2

Class Test

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

Total there are 7 identifiers.

Ex 3

Class Demo

```
{
    public static void main(String [args])
        { String name = "Sachin";
          String result = name.toUpperCase();
          System.out.println(result);
        }
}
```

Rules for writing an identifier:

1. The only allowed characters in java identifiers are:
a to z, A to Z, 0 to 9, -(underline), \$
2. If we use any other characters it would result in error.

3. Identifiers are not allowed to start with digits.

int telusko1 = 100; (valid)

int 1telusko = 100; (invalid)

4. Java identifiers are case sensitive (number != Number)

5. There is no length limit on java identifiers, but still it is a good practice to keep the length not more than 15 characters.

6. We can't use reserved words as identifiers

eg: int if = 10; (compiler error)

* String - inbuilt class name

* Runnable - inbuilt interface name.

* Student - user defined class name

7. Predefined class names can be used as identifiers

eg: * String Runnable = "Sachin";

System.out.println (Runnable); || Sachin.

* int String = 10;

System.out.println (String); || 10

Even though predefined class names can be used as identifiers, it is not a good practice to keep-

* int If = 10; if != IF.

System.out.print (If); valid ✓

* int-int = 10;

System.out.print (int); (E.

Note:

Literal: Any constant value which can be assigned to a variable is called literal.

int data = 10; Literal → 10

data → Variable name, int → datatype

Note: for boolean datatypes the only values allowed for variable is "true" and "false", other than this if we try to keep the values (any) it would result in "Compile Time Error".

Q1. Which of the following list contains only reserve word?

1. final, finally, finalize

Ans: finalize is not a reserve word, it is a method in Object class.

2. break, Continue, exit, return

Ans: exit is not a reserve word, it is a method in System class.

3. byte, Short, Integer, long

Ans: Integer is not a reserved word, it is a predefined class.

4. throw, throws, throws

Ans: throws is not a reserved word, it is a user-defined variable.

Datatypes:

Every variable has a type, every expression has a type and all types are strictly typed / defined in java, because java is strictly type / statically typed language.

Compiler role → Compiler will check the value stored can be handled by datatype or not. This checking which is done by compiler is called "Type checking / Strictly type checking"

Primitive Datatypes: data which is commonly used and supported by any language to store directly.

a) Numeric values: to store number

- whole number, real number

b) Character values: to store character type of data

c) Boolean values: to store logical values.

Number data:

To store whole numbers we have 4 datatypes

- a) byte
- b) short
- c) int
- d) long

Data-type information like:

- a) Size of datatype (how much memory is allocated on the ram for that datatype by Jvm)
- b) min value what it can keep
- c) max value it can keep.

Note:

- * `System.out.println ("Size of byte is :: "+Byte.SIZE);`
- * `System.out.println ("minvalue of byte is :: "+Byte.MIN_VALUE);`
- * `System.out.println ("Maxvalue of byte :: "+Byte.MAX_VALUE);`

a. byte:

Output: Size → 8 bits

min value → -128

max value → 127

eg:

`byte mark = 35 // valid`

`byte mark = 135 // CE: possible loss of precision`

`byte a = "a"; // CE: incompatible types.`

Q2 When to use byte datatype?

It is commonly used when we handle the data which is coming from Stream, network.

Stream → java.io package

" " → means String

' ' → char data

b.

Short :

Size : 16 bits (2 byte)

min value : -32768

max value : +32768

Note: this data is not at all used in java and this datatype is best suited only if you have old processors.

c.

int :

Size : 32 bits (4 byte)

min value : -2147483648

max value : +2147483647.

Note: the most commonly used datatype for storing whole number is "int" and by default if we specify any literal of number, compiler will keep it as "int".

d.

long :

Size : 64 bits (8 bytes)

min value : -2^{64-1} max value : $+2^{64-1}$

Note: when we work with large file, data would come to java programs in terms of GBs, int is not enough so we use long.

* If the data goes beyond the range of int, then keep the data inside the long data type we need to explicitly suffix the data with 'L' or 'l' otherwise it would result in "Compile Time Error".

Real Number

(a)

float

Size : 32 bits (4 byte)

min value : 1.4E-45

max value : 3.4028235E38

Note: by default if you specify any real number / dec number compiler will treat it as "double", to specify to treat as float, we need to suffix with "F" or "f".

eg: float a = 10.5 - C.E

float b = 10.5f; { valid ✓ }

float c = 25.5F;

b) double

Size: 64 bits (8 bytes)
 minValue: 4.9E-324
 maxValue: 1.79...E308

eg: double d = 23.567;

double d = 1.79...57E308;

Note: Datatypes are actually represented to the compiler and
 just using reserve words. reserve words normally is
 "lower case".

To map primitive data as Object in java from JDK 1.5 concept
 of "Wrapper Class" were introduced

- | | |
|------------------|--------------------|
| 1. byte → Byte | 4. long → Long |
| 2. Short → Short | 5. double → Double |
| 3. int → Integer | 6. float → Float |

Char Data type:

Size: 16bit (2 Byte)

Java was developed for multilingual languages it adopted
 the Unicode System i.e there are 65,535 characters
 whereas in 'C' char has just 8 bit (1 byte) and it adopted
ASCII System (0 to 128)

eg:

Char a = 'A'; **valid**

Char a = "A"; **invalid** - *Single quote only

Char a = "AB"; **invalid** - no two chars allowed, *only one!

* Char → Character (class)

Type Casting: Changing one type of data to another type.

→ (Implicitly) → without any efforts or automatically
 is called "implicit type casting".

eg: int a = 25;

int b = 2;

also called "Numeric type promotion"

float c = a/b; O/p = 12.0 (int → float converted)

* Implicit Typecasting happens in

byte → short → int → long → float → double

left to right is possible but reverse is not possible implicitly.

Explicit type casting : Changing type of data to another type with extra effort is called "explicit typecasting".

Eg: 1. double a = 45.5;

byte b;

b = (byte) a;

System.out.println(b); Output: 45 (Data loss may occur)

2. byte ab = 10;

byte ac = 20;

byte res = ab * ac;

* result in error

Solution:

* (byte) ab * ac;

= * int res = ab * ac;

because resultant will be

an integer by default.

Incrementation

* Pre increment = ++a;

* Post increment → a++;

Eg: int a = 5; int b;

b = a++;

System.out.print(a); - 6

System.out.print(b); - 5

If: b = ++a;

System.out.print(b); - 6

Decrementation

* Pre decrement → --a;

* Post decrement → a--;

* In pre increment or pre decrement the value of variable is first changed then assigned to a variable if assignment operator is there.

Example: int a = 5;
 int b;
 $b = a++ + a++ + ++a; \rightarrow 5 + 6 + 8$
 $\text{System.out.print}(a); \rightarrow 8$
 $\text{System.out.print}(b); \rightarrow 19$

Code Snippets:

1. For the code below what should be the name of java file?

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

Ans: **HelloWorld.java** (Identifiers are case sensitive)

2. public class myfile

```
{ public static void main(String [ ] args)
{
  String arg1 = args[1];
  String arg2 = args[2];
  String arg3 = args[3];
  System.out.println("Arg is " + arg3);
}
```

Which command line arguments should you pass to program
 to obtain the following output: Arg is 2

Ans: **java myfile 1 3 2 2.** \downarrow **arg[3]**, so 4th index = 2.

3. public class Test

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

Does the code compile successfully?
 \downarrow
 Ans: Yes

multiple Semicolons won't do anything.

4. What is the signature of special main method?

Ans: public static void main (String [] a)

5. What will be result of Compiling and executing Test class?

java Test good morning everyone

Private Class Test

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

Ans: Compilation error (because of private class)

6. For the Class Test, which options, if used to replace /* INSERT */,
will print "Hurrah I passed...." on to the console?

Public Class Test

```
{
    /* INSERT */
    {
        System.out.println ("Hurrah I passed..."); }
}
```

Ans: public static void main (String [] args)

static public void main (String [] a)

7. Suppose you have created a java file , "myClass.java".

Which of the following commands will compile java file?

Ans: javac MyClass.java

'for compilation and java for execution'

Operators in Java

1. Arithmetic Operators (+, -, *, /, %)
2. Increment and decremental operators (++ and --)
3. Logical operators (&&, ||, !)
4. Assignment operators (=)
5. Conditional operators (if, else, ternary operator)
6. Relational Operators (==, !=, >, <, >=, <=)
7. Bitwise and shift operators

* Unary Operator: Unary operators require only one operand.
 operations such as : incrementing / decrementing, negating an expression or inverting a boolean.

* Binary Operators: Operators which require atleast two operands to perform operations are called binary op.

* Ternary Operator: Operator which require 3 operands and it is the only conditional operator that takes three operands.

eg: int a=s, b=10;
 int res = (a>b)? a: b; (Syntax (exp1)? a:b)
 System.out.print(res); → 10

Switch Statement : The switch statement execute one statement from multiple conditions like if - else if ladder Statement.

Syntax: Switch (expression) {

Case Value 1 : Code to be executed ; break;

Case Value 2 : Code to be executed ; break;

default : Code to be executed if no case matching }

* break - optional.

Code Snippets:

1. `public class Test {
 public static void main(String[] args)
 { args[1] = "Day!";
 System.out.print(args[0] + " " + args[1]);
 }
}`

→ And the commands : `javac Test.java`

~~`java Test Good`~~

Result?

Ans: Jvm would create a problem during execution as `args[1]` there was no memory, memory was till `args[0]` because of one argument "Good" so .exception!

2. File name: Test.java

`public class Test {
 public static void main(String[] args)
 { System.out.println("Welcome" + args[0] + "!");
 }
}`

→ And the commands are : `javac Test.java`

What is the result?

~~`java Test "James Gosling" "Bill Joy"`~~
~~args[0]~~

Ans: Welcome James Gosling! (we use " " to ignore the space).

3. `public class Test {`

`public static void main(String[] args)
 { boolean b1=0;
 boolean b2=1;
 System.out.print(b1+b2);
 }
}`

Ans: Compilation Error.

4. Given:

35. String #name = "Jane Doe";

36. int \$age = 24;

37. Double -height = 123.5;

38. double ~temp = 37.5;

Which two statements are true?

Ans: Line 35 and 38 will not compile.

5. public class Test {

public static void main (String [] args) : public static void main

{ byte b1 = (byte) 127 + 21; } it's correct.

System.out.print(b1); } ↴ 148 can't be stored in byte

}

Ans: -108

so: Jvm will do minrange + (rec - max - 1)

= -128 + (148 - 127 - 1)

= -108 (formula for every datatype)

6. public class Test {

public static void main (String [] args)

{ char c1 = 'a'; // ASCII value of 'a' is 97 }

int ii = c1;

Char → int (implicit typecasting)

System.out.print(ii); }

}

Ans: 97

7. public class Test {

public static void main (String [] args)

{ byte b1 = 10;

int ii = b1; byte → int (implicit)

byte b2 = ii; int → byte (not possible, need to tell explicitly)

System.out.print (b1 + ii + b2); }

}

Ans: Line 3 causes compilation error.

8. What will be the output?

```
int x = 100;
```

```
int a = x++; → a=100, x=101
```

```
int b = ++x; → b=102, x=102
```

```
int c = x++; → C=102, x=103.
```

```
int d = (a < b) ? (a < c) ? a : (b < c) ? b : c : x;  
(100 < 102) ? (100 < 102) : 100
```

```
System.out.print(d);
```

Ans: The output will be 100.

9. Class Test {

```
public static void main (String [ ] args)  
{ int a = 100;  
    System.out.print (-a++); }
```

Ans: -100.

Loops in Java: Looping in programming language is a feature which facilitated the execution of set of instructions repeatedly while some condition is true.

Types of loops are: ① for ()

② while ()

③ do-while ()

④ for-each (enhanced for loop)

1. for



Initialization



Condition check



(Yes)



Body of loop



Update

No

→ for (init; cond^{!=}; update)



Body

e.g. for (int i=0; i<n; i++)

{ System.out.print ("*"); }

2. while (boolean condition)
 { loop statements... }

While loop starts with checking boolean condition. If true executed otherwise not. It is also called as "Entry control loop".

3. do while

do {

 Statements }

while (condition);

do while loop is similar to while loop with only difference that is it checks for condition after executing statements, and therefore also called as "Exit control loop".

Ex. 1. int n=4

for (int i=0; i<n; i++)

{ for (int j=0; j<n-1; j++)

{

 if (i==0 || i==3 || j==0 || j==3)

 System.out.print ("*");

 else

 System.out.print (" ");

Output:

 j=0 1 2 3,

 i=0 * * * *

 1 * * *

 2 * * *

 3 * * * *

System.out.println();

}

Ex. 2. int n=10;

for (int i=0; i<n; i++)

{ for (int j=0; j<n; j++)

{ if ((i==0 && j<n-1) || j==0 || (i==n-1 && j<n-1) ||

(j==n-1 && i>0 && i<n-1))

 System.out.print ("*");

O/P: * * * * * "D".

 else System.out.print (" ");

}

Code Snippets :

1. Public Class Test

```
{
    public static void main (String [] args)
    {
        char c = 'z';
        long l = 100_001;
        int i = 9_2;
        float f = 2.02f;
        double d = 10_0.35d;
    }
}
```

$l = c + i; \rightarrow \text{char} + \text{int} = \text{int} \rightarrow \underline{\text{long}}$
 $f = c * l * i * f; \rightarrow \text{float} (\text{higher datatype})$
 $f = l + i + c; \rightarrow \text{long} \rightarrow \text{float}$
 $i = (\text{int}) d; \rightarrow \text{long} \rightarrow \text{float}$
 $f = (\text{long}) d; \quad ? \quad (\text{Implicit})$

Does code compile successfully?

Ans: YES.

Note: from Java 1.7 for a literal we can give '-' also, if we give compiler will remove that '-' in .class file.

2.

```
{
    int a = 20;
```

```
    int var = --a * a++ + a-- - a--a;
```

```
    System.out.print ("a=" + a);
```

```
    System.out.print ("var=" + var); }
```

Result : a= 18 var= 363

3.

```
{
    int i=5;
```

```
    if (i++ < 6) { 5 < 6
```

```
        System.out.println (i++); }
```

(6 then ++)

Output: 6

4.

```
int x=4; int y= 4++;
```

Ans: Compilation error.

```
System.out.println (x);
```

```
System.out.println (y);
```

Post increment done only on variables not literals.

5. `int x = 4;`

`int y = ++(++x);` → Line 2 : Compile Time Error

`System.out.print(x);` (Because incrementation done only on variables not on direct literals).

6. `boolean b = true;`

`b++;` Ans: Line 2 : Compile time error

`System.out.println(b);` because increment/decrement not applicable on boolean type.

7. `int b, c, d;`

`int a = b = c = d = 10;` Ans: Yes, the code will compile.
Will the code compile?

8. `int a = b = c = d = 20;`

Ans: No, because b, c, d not initialized.

`System.out.println(a);`

9. `byte c = (10 > 20) ? 30 : 40;` int

`byte d = (10 < 20) ? 30 : 40;` Ans: 40

`System.out.println(c);` 30

`System.out.println(d);`

10. `int a = 10;` int `b = 20;` - type checking is valid

`byte c = (a > b) ? 30 : 40;` - literals are not involved in operation,

`byte d = (a < b) ? 30 : 40;` So compiler would just check typechecking of result.

Ans: Compile Time Error.

11.

`int x = 5 --- 0;`

How many statements are legal?

`int y = --- 50; ^`

'Should not be in beginning/end, okay in b/w.

`int z = 50 ---; ^`

Ans: three statements only

`float f = 123.76 - 86f;`

`double d = 1 - 2 - 3 - 4;`

13. `int x = 10;`

`if (++x < 10 && (x/10) > 10) {`

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

`else`

`System.out.println("Hi");`

Ans: "Hi" is printed.

13.

`int i = 10;`

`int j = 20;`

Ans: 10:30:6

`int k = (j + i) / 5;`

`System.out.println(i + ":" + j + ":" + k);`

14.

`int x = 10;`

`if (x) System.out.println("Hello");`

`if (boolean exp.) = Syntax`

`else System.out.println("Hiee");`

`in this code int is found
so error.`

Ans: Compile Time Error.

15.

`int x = 10;`

`if (x = 20) System.out.println("Hello");`

`else System.out.println("Hiee");`

Ans: Compile time error because of assignment operator.

16.

`boolean b = false;`

`if (b = true) System.out.println("Hello");`

`else System.out.println("Hiee");`

Ans: "Hello" because assignment operator is evaluated on boolean type.

17.

`if (true)`

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

Ans: "Hello" is printed

18. Public class Test {

 public static void main (String [] args) {

 if (true); }

}

Ans: No output. (";" is a valid java Statement)

19. Public class Test {

 public static void main (String [] args) {

 if (true)

 int x=10; }

}

Ans: Compile time error.

Note: if there is only one statement which needs to be a part of "if" then {} is optional, but statement should not be a declarative statement.

20. if (true) {
 System.out.println ("hello");
 System.out.println ("bye");}

How many statements are independent of if?

Ans: 1 statement.

OOPS

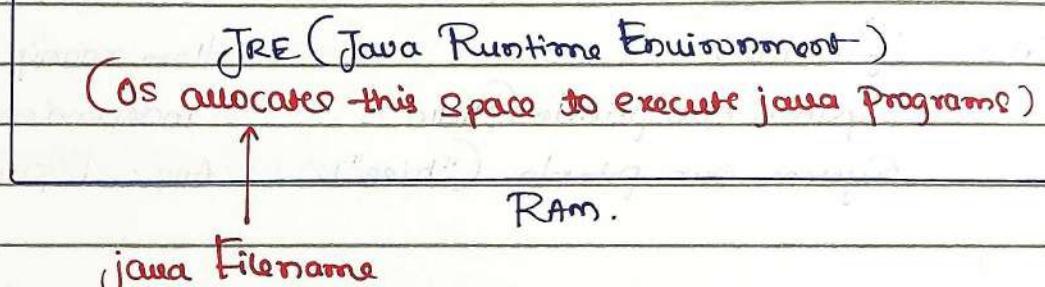
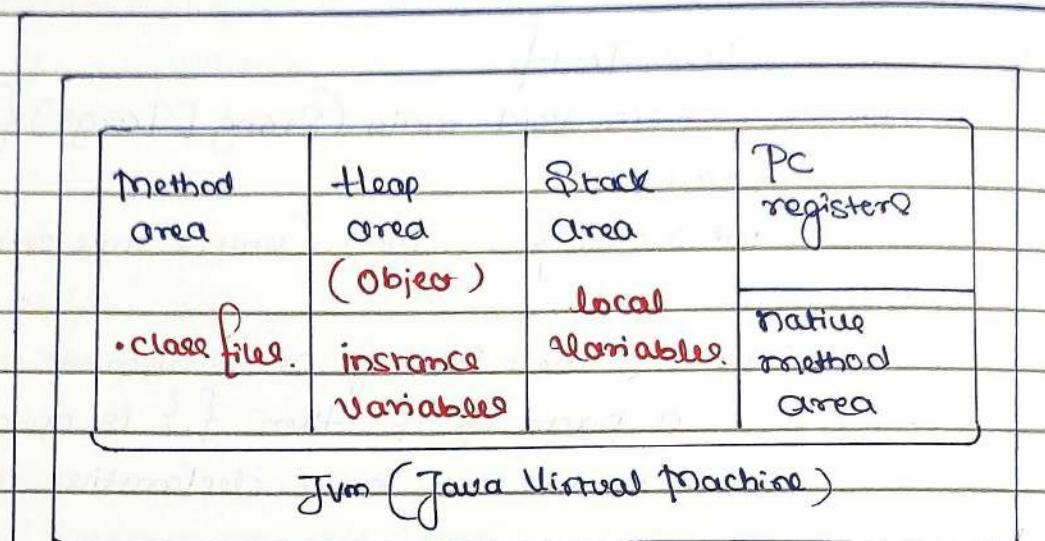
It is actually theory concept, which is implemented by many programming language like C++, java, python.... Any real time problem can be solved if we solve oops principle

To OOPS while solving the problem

1. We need to first mark the objects
2. Every object we mark should have 2 parts:
 - a. HAS-part / attributes (store info. as variables)
 - b. DOES-part / behaviours (represent them as methods)
3. To represent an object, first we need to have a blueprint of an object.
4. We use "new" keyword / reserve word to create an object for blueprint.

5. Every object should always be in constant interaction
 6. Useless object doesn't exist.

JVM architecture:



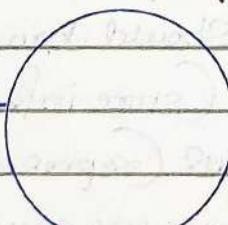
PC registers: address of next instruction which needs to be executed

Native method area: code of other languages which is required for java would be available here.

Student std = new Student();

↓
123457ACEF (HashCode of the Object)

Std



Student

(Heap Area)

1. What is Object?

Physical existence of any element we say as object.

e.g.: book, car, computer...

2. What is "Has-part" and "Does-part" of an object represents?

Has part : indicate what it can hold.

Does part : indicate what it can do

3. What is Blueprint in Java and how to represent it?

In Java to represent a blueprint we have a reserve word called "class".

Conventions followed by Java developer while writing a class is:

a.) Classname should be in "Pascal Convention".

e.g.: InputStream, FileReader...

b.) Variables are represented in "CamelCase".

e.g.: javaFullStack.

c.) methods are represented in "camel case".

e.g.: // Blueprint of Student object

Class Student

{ // HAS-part

 int Sid;

 String name;

 int age;

 char gender;

// DOES-part

 void play() { }

 void study() { }

}

* To create an object in java
we use "new" keyword.

Syntax:

Classname Variable = new

classname();

new:

it is a signal to jvm to
Create some space for object in
heap area.

new: It is a signal to Jvm to create some space for the object in the heap area.

- * Tell the className, JVM informs the className, JVM creates the object and sends the "hashcode" to the user.
- * User should collect the hashcode through "ref. variable"

Types of Variables:

Division 1 - Based on the type of value represented by a variable all variables are divided into 2 types:

1. Primitive Variable: primitive Variable used to represent primitive values. eg: int x=10;
2. Reference Variable: reference Variable can be used to refer objects.
eg: Student s = new Student();

Instance Variables: if the variable is declared inside the class, but outside the methods such variables are called as "instance variables".

or

If the value of the variable changes from object to object then such variables are called as "instance variables".

When will the memory for instance variable be given?

Ans: Only when the object is created JVM will create a memory and by default JVM will also assign the default value based on the datatype of variable.

eg: int - 0, float - 0.0f, boolean - false, char - ' ', String - null ...

Note: Scope of the instance variable would be available only when we have reference pointing to the object, if the object reference becomes null, then we can't access "instance variables".

eg. public class Test {

 int i=10;

 public static void main(String[] args)

 { System.out.println(i); } // CE: because instance variable can't be accessed directly in static context.

Test t = new Test(); // Object stored in heap area.

System.out.println(t.i);

t.methodOne(); }

public void methodOne()

{ // Inside instance method instance variable can be directly accessed.

System.out.println(i); } // 10 because it is an instance variable

}

}

Key points about instance variables:

- * If the value of a variable is varied from object to object such type of variable are called instance variable.
- * For every object a separate copy of instance variable will be created.
- * Instance Variables will be created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is same as scope of objects.
- * Instance Variables will be stored on heap as a part of object.
- * Instance Variables will be declared within the class directly but outside of any method or block or constructor.

- * Instance variables can be accessed directly from instance area but cannot be directly accessed from static area.
- * But by object reference we can access instance variables from static area.

Local Variables:

1. Variables which are created inside the stack area are called local variables.
2. During the execution of the method the memory for local variables will be given, and after the execution of the method the memory for variables will be taken out from the stack.
3. Local variables default value will not be given by the JVM, programmer should give the default value.
4. If the programmer doesn't give default value and if he uses the variable inside the method then program would result in "CE".

eg: public class Test

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

```
{ int i=0;
```

```
for (int j=0; j<3; j++) {
```

```
i = i+j;
```

```
}
```

```
System.out.println(j); } } || CE: 'j' not declared
```

eg. 2. Class Test

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

```
{ int a:
```

```
? System.out.print ("hello"); } } || 'hello' because 'a' not used.
```

eg 3 Class Test

```
{
    public static void main (String [large])
    {
        int x;
        System.out.println(x); } // CE: 'x' not initialised
    }
```

eg 4. Class Test

```
{
    public static void main (String [large])
    {
        int x;
        if (large.length > 0) x=10;
        System.out.println(x); }
    }
```

Keypoints of Local Variables:

- * Sometimes to meet temporary requirements of the programmer we can declare variable inside a method or block or constructor. Such type of variable are called local variables or automatic variables / temporary variable / stack variable.
- * Local variables will be stored inside stack.
- * The local variables will be created as part of the block execution in which it is declared and destroyed once the block execution completes. Hence scope of local variables is exactly same as scope of the block in which we declared.
- * It is highly recommended to perform initialization for the local variable at the time of declaration atleast with default values.

Code Snippets:

1. public class Test

{ public static void main(String args[])

{ int x=10;

switch (x)

{ System.out.println("hello"); }

{ } } ↓ Statement is not part of case label so CTE.

{ }

2. public class Test

{ public static void main(String args[])

{ int x=10;

int y=20;

switch (x)

{ Case 10: System.out.println("hello"); }

break;

Case y: System.out.println("hiee");

break; }

{ }

Ans: Compile time error because case label is not constant

* Label in switch should be "compile time constants", meaning the value should be known to compiler.

3. In the above code ↑

int x=10;

Ans: "hello".

final int y=20;

switch (x)

{ Case 10: System.out.println("hello"); }

break;

Case y: System.out.println("hiee"); ??

Note: "final" means compiler will get to know the value and compiler treats it as "Compile Time constant".

4. { int x = 10;
 Switch (x+1)
 { Case 10:
 Case 10+20;
 Case 10+20+30: } }

Ans: No output.

5. byte x = 10;
 Switch (x)
 { Case 10: System.out.println ("Hello"); **Ans: Compile time error because**
 break;
 Case 100: System.out.println ("Free"); **core value outside the range**
 break;
 Case 1000: System.out.println ("Bye"); **the range**
 break; }

Note: Label value should be within the range of `byte` type
 otherwise it would result in "CE".

6. byte x = 10;
 Switch (x+1) || `byte + int → int`, so `Switch(int)`
 { Case 10: System.out.println ("Hi");
 break;
 Case 100: System.out.println ("Hi"); **Ans: No output.**
 break;
 Case 1000: System.out.println ("No");
 break; }

7.

```

int x = 97;
switch(x)
{
    case 97 : Print("hi"); break;
    case 'a' : Print("100"); break;
}
    
```

' int x='a' x=97'

And case label value can't be duplicated so "CE".

Methods in Java:

A method in Java is a block of code that when called performs specific actions mentioned in it.

Methods have 4 things:

- ① name
- ② input (parameters)
- ③ Body
- ④ return type.

In my program if I have any task, I will write inside method.

Task is present inside a method and if the task has to be executed it has to be brought into stack area. What will be there in Stack area only that part will be executed.

Class Calculator

```

{
    int a,b,c; // Instance var.
    void add()
    {
        a=10;
        b=20;
        c=a+b;
        System.out.println(c);
    }
}
    
```

Public Class Launch

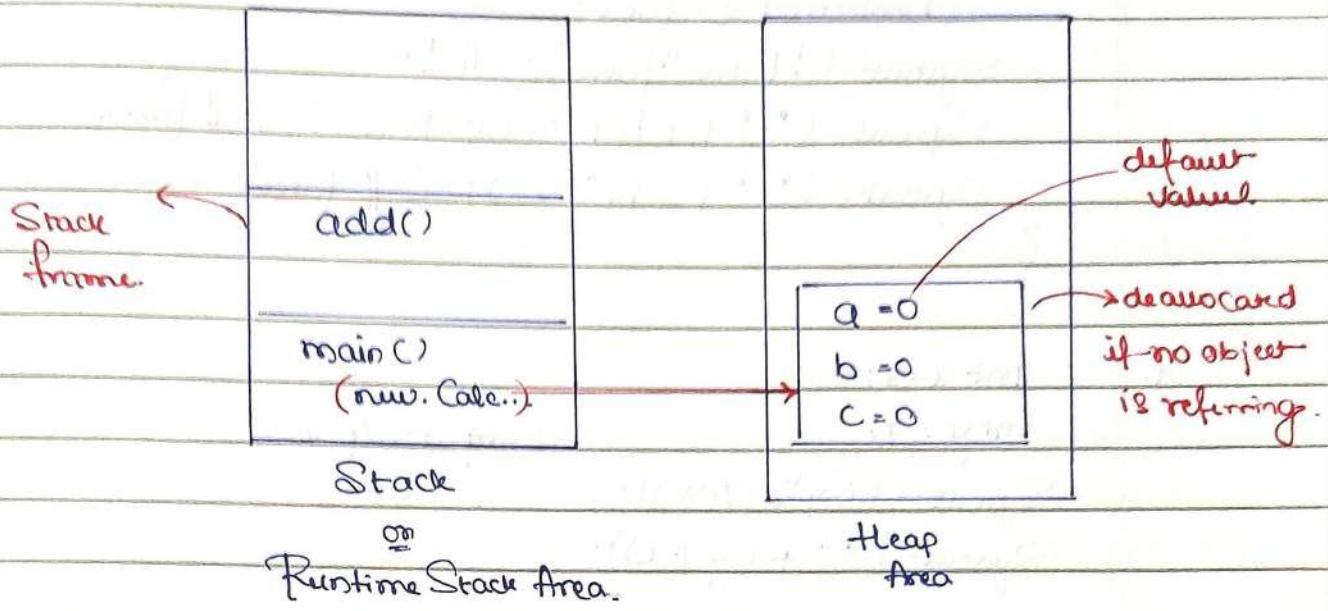
```

{
    public static void main (String []
    {
        ...
    }
}
    
```

Calculator calc = new Calculator();

calc.add();

Calling method.



Code Snippets:

1. int x = ?

Note: replace x with 0, 1, 2, 3.

Switch(x)

```
{
    default : Print("default");
    Case 0 : Print("0"); break;
    Case 1 : Print("1");
    Case 2 : Print("2");
}
```

x=0, out: 0

x=1 out: $\frac{1}{2}$

x=2 out: 2

x=3 out: default

.

2. Boolean b1=true; //Wrapper Class.

boolean b2=false;

boolean b3=true;

if ((b1&b2) | (b2 & b3) & b3)

Ans: No output is produced

System.out.print("alpha");

if ((b1=false) | (b1&b3) | (b1|b2))

System.out.print("beta");

3. class Maybe

```
{
    public static void main (String [ ] args)
    {
        boolean b1 = true;
    }
}
```

```

boolean b2 = false;
System.out (!false ^ false); // true
System.out (" " + (!b1 & (b2=true))); // false
System.out. (" " + (b1 ^ b2)); } // false
}

```

4.

```

int x=5;
int y=7;
System.out (((y+2) % x));
System.out (" " + (y % x));

```

Output: 4 2

5.

```

Integer i = 42;
String s = (i<40) ? "life" : (i>50) ? "universe" : "everything";
System.out (s);

```

Ans: "everything".

6.

```

{ Integer x=0;
  Integer y=0;
  for (Short z=0; z<5; z++)
    if ((++x>2) || (++y>2)) Ans: 82
      x++;
  System.out (x + " " + y); }

```

Method Overloading

Method Overloading in java is a feature that allows a class to have more than one method with the same name, but with different parameters.

Java supports method overloading through two mechanisms:

- ① By changing the number of parameters
- ② By changing the datatypes of parameters.

eg: Class Calculator

```
{  
    int add (int a, int b)  
    { return a+b; }
```

```
int add (int a, int b, int c)  
{ return a+b+c; }
```

```
float add (int a, float b)  
{ return a+b; }
```

void main ()

```
{  
    Calculator calc
```

```
= new Calculator();
```

```
int a=10, b=20, c=30;
```

```
float m=5.5;
```

```
Sysout (calc.add (10, 20));
```

```
Sysout (calc.add (10, 20, 30));
```

```
Sysout (calc.add (20, a, m));
```

1. Can we overload main method in Java?

Ans Yes, we can overload main method however Jvm will call such a main method which accept "String [] args" as parameters.

eg: public class LaunchMain

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

```
public static void main (int [] args)  
{ System.out.println ("accepting args"); }
```

Output: "Yes".

Arrays in Java:

* Arrays are index based data structure to store

large volume of data using single name (variable name)

* Arrays store homogeneous type of data

* Arrays are treated as objects and allocated on heap.

Syntax :

`int [] a = new int[10];` → 1D Array

'a' is an array of '10' integers.

`int [] arr = new int[N];`

`int [][] arr = new int[3][4];` → 2D Array

* Jagged Array : Array where data is irregular.

eg:	Classes.	Students	{	int [][] arr = new int[3][];
	0	5		arr[0] = new int[5];
	1	3		arr[1] = new int[3];
	2	4.		arr[2] = new int[4]; ?

Code Snippets :

1. `public static void main (String [] args)`
`{ while (true); }`

Ans: Infinite loop with no output

2. `{ while (true)`
`int x = 10; }` Ans: CE because declarative statement
 are not allowed.

3. `while (true)`

`{ int x = 10; }` Ans: Memory for x will be given 4 bytes
 during execution.

4. `while (true) { Sysow ("Hello"); }`

`Sysow ("Hi");`

Ans: Unreachable code
 (Compile time error).

5. while (false)

```
{ System.out("hello"); }
```

```
System.out("hi");
```

Ans: Compile Time Error at line 1.
(unreachable code).

6. { int a=10, b=20;

```
while (a < b)
```

```
{ System.out("hello"); }
```

```
System.out("hi"); }
```

Ans: "Hello" will be printed infinite times

Note: Whenever variables are marked as final, compiler does not do know the value of those variables and it will use the value in the expression to get the result.

7. { final int a=10, b=20;

```
while (a < b)
```

Ans: Compile time error at line n2

```
{ System.out("hello"); }
```

```
System.out("hi"); }
```

8. { do

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

Ans: Hello infinite times.

```
while (true); }
```

9. { do

```
while (true); }
```

Ans: Compile Time Error (there should be atleast one statement in loop)

10. { do;

```
while (true); }
```

Ans: No output, Semicolon is an empty statement.

11. { do while (true)

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

Ans: Hello infinite times.

```
while (true); }
```

Taking input from Console:

```
Scanner Scan = new Scanner (System.in);
int a = Scan.nextInt();
(To take integer input)
```

Taking input in an array:

```
int[] ar = new int[s];
Scanner Scan = new Scanner (System.in);
for (int i=0; i<n; i++)
{ ar[i] = Scan.nextInt(); }
```

```
for (int i=0; i<s; i++)
{ System.out.println (ar[i]); }
```

Making an array of objects

```
Class Fan
{
    int cost;
    String brand;
    int noOfWings;
```

```
public class Launch
{
    public static void main (String [args])
    {
        Fan [1] f = new Fan[3];
        f[0] = new Fan();
        f[1] = new Fan();
        f[0].brand = "Viwell"; }
```

Disadvantages of Array:

- * It can store only homogeneous type of data.
 - * Memory of array is fixed in size, it cannot grow or shrink.
 - * Array demands contiguous memory locations.

Enhanced | Advanced "for Loop"

eq. int arr[] = { 10, 20, 30, 40 };

for (int x: arr)

{ System.out.println(x); }.

Output: 10

20

30

40

Code Snippets:

1. do {
 Sysout ("Hello"); } Ans: Compile time error at line 2.

whele (true);
Sysow ("hi");

Ans: Compile time error at line 2.

```
2. do {  
    Sysout ("Hello");  
    while (false);  
    Sysout ("Hi");  
}
```

Ans : huu

10

3. int i=0, j=0;
int i=0, Boolean
int i=0, int j=0;

How many statements are valid?

Ans: Only 1 statement is valid
(first). .

4. int i=0;

Ans : hello

```
for (Sysout("hello"); i<3; i++)  
{ Sysout("no"); }
```

Note: Syntax of for loop

```
for (Stmt1; Stmt2; Stmt3)
    { Stmt4; }
```

Stmt 1: Can be any statement, but suggested for initialization.

Stmt 2: Compulsorily should be a Boolean statement only.

Stmt 3: Can be any statement, but suggested for inc/dec.

Stmt 4: Can be any statement, suggested for repetitive logic.

5. `int i=0;`

```
for (Sysout("Hello"); i<3; Sysout("hi"))
    { i++; }
```

Ans: hello

hi x3.

6. `for (; ;)`

```
{ Sysout("Hello"); }
```

Ans: "Hello" infinite times

Boolean value / statement will be evaluated as true if empty.

7. `for (int i=0; true; i++)`

```
{ Sysout("Hello"); } Ans: Compile time error at line 2
```

`Sysout("Second");`

8. `for (int i=0; false; i++)`

```
{ Sysout("Hello"); }
```

Ans: Compile time error at line 1.
(code unreachable)

Note: * If a variable marked as final, then those values are known to compiler so we say them as "Compile-time constants".

* If a variable is marked as final, then the value for those variables should never be changed in the program, if we try to do will result in "Compile time error".

* In Java memory for a variable is given by JVM as per its datatype specification and value also will be assigned by JVM only. Compiler will not allocate memory for the variable and it will not initialize the value for the variable.

e.g:

```
final int a=10;
```

```
a++;    || CE: Value can't be reassigned.
```

```
System.out(a);
```

Enhanced for loop for 2D arrays.

e.g. `int[][] a = {{10, 20}, {30, 40, 50}, {60, 70, 80, 90}};`

```
for (int ar[] : a)
{
    for (int elem : ar)
        System.out.print(elem + " ");
}
```

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

Output: 10 20
30 40 50
60 70 80 90

Limitations of Enhanced for loop:

1. We cannot access a particular element at index and modify it.
2. Traverse iterates only in forward direction.
3. We cannot traverse alternative index or skip certain index.

Syntaxes for arrays:

`int []x; ✓`

`int []x; ✓`

`int [6]ar; ✗`

`x = new int[4]; ✓`

`int x[]; ✓`

`int [7]ar[]; ✓`

`int []aa, []bb; ✗`

`int []aa, bb[][], ✓ aa=10, bb=30`

Note :Size can be :

`int []a = new int [s];` → byte, short, int, char ✓
 ↓
 size. → long, float, double, boolean ✗

- * Range in an array is upto the range of "int" not beyond it.
- * There are many methods in Array class like sort, fill, binary-search etc.
- * All these methods are static methods which means we do not need to create objects to call methods, we can call the methods directly.

eg: `Arrays.sort();`

`int arr [] = {70, 80, 90};`

`Arrays.sort(arr);` // Sorts the array.

Code Snippets

1. `int x=0;`

`switch(x){`

`Case 0 : System.out.println("hello"); break; Ans: hello`

`Case 1 : System.out.println("hi"); }`

2. `for (int i=0; i<10; i++)`

`{ if (i==5) break; Ans: 0 1 2 3 4 }`

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

`}`

3. `int x=10;`

`if (x < 10) { System.out.println("begin");`

`if (x == 10) break; }`

`System.out.println("end"); }`

`System.out.println("hello");`

`Ans: begin`

`hello`

4. `int x=10;`

`if (x==10) break;`
`Sysout ("hello");`

Ans: Compile time error because break is used in non switch loop.

5. `int x=2;`

`for (int i=0; i<10; i++)`
`{ if (i*x==0) continue;`
`Sysout (i); }`

Ans: 1 3 5 7 9

6. `int x=10;`

`if (x==10) continue;`
`Sysout ("hello");`

Ans: Compile time error.

7. `int x=0;`

`Switch (x)`

`{ Case 0: Sysout ("Hello"); Continue;`
`Case 1: Sysout ("hi"); }`

Ans: Compile time error

- continue can be only used in loops and labelled blocks.

8. `if (true) Sysout ("Hello");`

`else Sysout ("hi");`

Ans: Hello.

Concept of unreachability holds good only for loops.

Binary Search

Binary Search is one of searching techniques applied when input is sorted

public class BinarySearch

{ public static void main (String [] args)

{ int arr [] = { 10, 20, 30, 40, 50 };

Scanner Scan = new Scanner (System.in);

System.out.println ("Enter the key");

```

int key = Scan.nextInt();
int low = 0;
int high = arr.length - 1;
while (low <= high)
{
    int mid = (low + high) / 2;
    if (key == arr[mid])
    {
        System.out.println("Key found");
        break;
    }
    else if (key < arr[mid]) high = mid - 1;
    else if (key > arr[mid]) low = mid + 1;
}
if (low > high) System.out.print("Key found");
    
```

Array Class

public class AC

```

    {
        public static void main (String [] args)
        {
            int [] a = new int [4];
        }
    }
    
```

Arrays.fill (a, 5); *// Fills '5' in every index*

Arrays.fill (a, 2, 5, 9); *// Fills '9' from index 2-4*

Arrays.sort (a); *// Sorts the array*

int res = Arrays.binarySearch (arr, 10);

// Searched for '10' in the sorted array & returns index, if not found returns negative index.

}

Code Snippets

1. int $x=0$;

def

$++x$;

Ans: 1 4 8 6 10

Sysout(x);

if ($++x < 5$) continue;

$++x$;

Sysout(x); }

while ($++x < 10$):

2. for (int $i=0$; $i++ < 5$; $i++$) *How many times hello printed?

{ Sysout ("hello");

$i = i++$; }

Ans: 3 times

assignment and increment to same variable, increment
does not have any scope, so increment doesn't happen.

3. int $i=5$;

Sysout ($i++$ " ");

Ans: 5 6 7 16

Sysout (i " ");

Sysout ($++i$ " ");

Sysout ($++i + i++$ " ");

4. int $[7]a = \{0, 2, 4, 1, 3\}$;

for (int $i=0$; $i < a.length$; $i++$)

Ans: 2

$a[i] = a[(a[i]+3)/a.length]$;

Sysout ($a[i]$);

5. int $i=0$; $j=5$;

for (; ($i < 3$) and ($j > 10$) ;)

Ans: 0 6 1 7 2 8 3 8

{ Sysout (" " + i + " " + j); $i++$; }

Sysout (i + " " + j);

6. for (int i=0; i<10; i+=2);
 System.out.println(i); Ans: 02468

7. $\text{int } x = 5;$
 $x^* = 3^*5 + 7^*x-1 + \text{++}x;$ Ans: 275
 $\text{System}(x);$ ↑ assignment an

8. `int a = 3;` no Scope.
`switch(a)`

{ Case 1: `++a`; ans: 5

Case 2: att;

Case 3: `a++`;

default : ++a; ?

Sysout(a);

Assignment and incrementation to the same variable, so incrementation has no scope.

9. `int i;
for (i=0; i<6; i++)
{ if (++i>3) continue;
 System.out.println(i);}`

Bubble Sort

Public Class LaunchES

{ Public static void main (String [large])

```
{  
    int []a={7,5,2,3,1,4,6};
```

```
for (int i=0; i<a.length; i++)
```

```

    { for (int j=1; j<a.length-i; j++); } } pushing the large
    { if (a[j] < a[j-1]) element to last
        swap(a[j], a[j-1]); } } by swapping.
    }
}

```

Guesser Game Assignment.

```
import java.util.*;  
class Guesser  
{ int guessNum;  
    int guessNum()  
    { Scanner Scan = new Scanner (System.in);  
        System.out.println ("Guess the numbers");  
        guessNum = Scan.nextInt();  
        return guessNum; }  
}
```

Class Player

```
{ int guessNum;  
    int guessNum()  
    { Scanner Scan = new Scanner (System.in);  
        System.out.println ("Player's Guess");  
        guessNum = Scan.nextInt();  
        return guessNum; }  
}
```

Class Vampire

```
{ int numFromGuesser;  
    int numFromPlayer1;  
    int numFromPlayer2;
```

void CollectNumFromGuesser()

```
{ Guesser g = new Guesser();  
    numFromGuesser = g.guessNum(); }
```

void CollectNumFromPlayers()

```
{ Player p1 = new Player(); }
```

```
Player p2 = new Player();  
numFromPlayer1 = p1.guessNum();  
numFromPlayer2 = p2.guessNum(); }
```

Void compare()

```
{ if (numFromGuesser == numFromPlayer1)  
{ if (numFromGuesser == numFromPlayer1 && 2)  
    System.out.println ("All players won");  
  
else if (numFromGuesser == numFromPlayer2)  
    System.out.println ("Player 1 and 2 won");  
  
else if (numFromGuesser == numFromPlayer3)  
    System.out.println ("Player 1 and 3 won");  
  
else  
    System.out.println ("Player 1 Won"); }  
  
else if (numFromGuesser == numFromPlayer2)  
{ if (numFromGuesser == numFromPlayer3)  
    System.out.println ("Player 2 and 3 won");  
  
else  
    System.out.println ("Player 2 Won"); }  
  
else if (numFromGuesser == numFromPlayer3)  
{ System.out.println ("Player 3 Won"); }}
```

else

```
{
    System.out.println("Game Lost, Try Again!"); }
```

Public Class LaunchGame

{

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

{

```
Vampire v = new Vampire();
```

```
v. CollectNumFromGuesser();
```

```
v. CollectNumFromPlayers();
```

```
v. Compare(); }
```

{

String in Java

- * It is a class name, String is basically an immutable class for which object can be created.
- * String refers to collection of characters.

Eg: Syntax:

```
String s1 = "Sachin";
```

```
System.out.println(s1);
```

* String class and
without "new".

```
String s2 = new String("Sachin");
```

```

graph TD
    String --> Immutable
    String --> Mutable
    
```

Immutable
(no change)

Mutable (change)

1. StringBuffer

2. StringBuilder (1.5v)

1. String (c)

* In Java String object is by default immutable, meaning once the object is created we cannot change the value of the object, if we try to change then those changes will be reflected on the new object not on the existing object.

Case 1: `String s = "Sachin";`

`s.concat("tendulkar");` || new object of string got created in heap area "Sachintendulkar" so immutable, the new object gets collected by garbage collector.

`System.out.println(s);`

Output: Sachin

VS

`StringBuilder sb = new StringBuilder("Sachin");`

`sb.append("tendulkar");` || on same object modification so mutable.

`System.out.println(sb);`

Output: Sachintendulkar

Case 2:

`String s1 = "Sachin";`

`String s2 = new String("Sachin");`

`System.out.println(s1 == s2);` || False because they are two different objects.

`System.out.println(s1.equals(s2));` || true.

String class.equals method will compare the content of the object, if returns true if same else false.

V/S

String Builder s1 = new StringBuilder("sachin");

String Builder s2 = new StringBuilder ("sachin");

System.out.println(s1==s2); // False

System.out.println(s1.equals(s2)); // False

String Builder class equals() compare the reference
(Address of object) not the Content of String Builder

Case 3:

String s1 = new String("sachin");

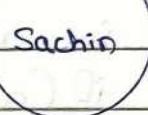
In this case two objects
will be created one in heap
and another one in String
Constant Pool, the reference
will always point to heap.

v/s

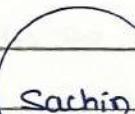
Stack

s1 →

Heap Area



String ConstantPool (SCP)



Garbage
Object
but can't
be collected.

String s="Sachin"

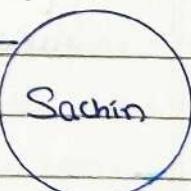
In this case only one
object will be created in
the String Constant Pool and
it will be referred by our
reference.

Stack

s1 →

Heap Area

String ConstantPool (SCP)



Code Snippets

1. int a=7;

boolean res = a++ == 7 && ++a == 9 || a++ == 9;

Sysout ("a = " + a);

Sysout ("res = " + res); Ans: a=9, res=True.

2. if (args.length == 1 | args[1].equals("test"))

Sysout ("test case");

else

Sysout ("production" + args[0]);

Arguments: java Fork 1me2

Ans: Exception thrown at runtime (array index out of bounds).

3. int aVar = 9;

if (aVar++ < 10)

System.out.print (aVar + "Hello World!"); Ans: 10HelloWorld!

else

System.out.print (aVar + "Hello Universe!");

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

for (x)

{ System.out.println (a[x]); }

→ What can u replace with x to print 135?

Ans: int e=0; e<5; e+=2;

5. int num=5;

do

{ System.out.println (num-- + ""); }

while (num==0);

Ans: 5

6. `int ii = 0;
int jj = 7;
for (int ii = 0; ii < jj - 1; ii = ii + 2;) { Sysout(ii); }`

Ans: 0,2,4.

7. `arr[0] = 10;
arr[1] = 20;
Sysout(arr[0] + ":" + arr[1]);`

→ Which code fragment should be kept first to print 10:20?

Ans: int arr;

`arr = new int[2];`

8. `boolean opt = true;
switch(opt) {
 case true: Sysout("tree");
 break;
 default: Sysout(" ***"); }
Sysout("Done");`

Note: Switch (args.)

args can be: byte, short, int,
char, String, enum.

9. `int intArr[] = {15, 30, 45, 60, 75};
int Arr[2] = intArr[4];
int Arr[4] = 90;`

What are values of each element in the array?

Ans: 15 30 75 60 90

10. `int x;
Sysout("Hello");`

Ans: Hello.

11. `int x;
Sysout(x);`

Ans: Compile time error (x is not initialised).

12. `int x;
if (args.length > 0) x = 10;
Sysout(x);`

Ans: Compile time error, because in case if condition fails then what would be the value of x? so JVM will give error.

Strings (Continued)

eg1.

```
String s1 = new String("dhoni");
```

```
String s2 = new String("dhoni");
```

```
System.out.println(s1 == s2); // False
```

```
String s3 = "dhoni";
```

```
String s4 = "dhoni";
```

```
System.out.println(s3 == s4); // True
```

Note: * Duplicate allowed in Heap Area
and not in SCP

* Memory will be cleaned at
the time of JVM shutdown.

Stack

s1 →

Heap Area

dhoni

s2 →

dhoni

SCP

s3 →

dhoni

s4 →

dhoni

eg2

```
String s = new String("sachin");
```

```
s.concat("tendulkar");
```

```
s = s.concat("IND");
```

```
s = "Sachintendulkar";
```

```
System.out.println(s); // Sachintendulkar
```

Stack

s →

Heap Area

Sachin

X Sachin tendulkar

s →

SCP

Sachin IND

Note: Direct Literals are always placed in SCP, because of runtime operation if object is required to create compulsorily that Object should be placed on heap, but not on SCP.

S →

Heap Area

Sachin tendulkar

X tendulkar

SCP

Sachin

IND

eg.3
 String s1 = new String ("Sachin");
 s1.concat ("tendulkar");
 s1 += "IND";

Stack.

Heap Area

s1

~~Sachin~~

String s2 = s1.concat ("MI");

System.out.println (s1); // SachinIND

System.out.println (s2); // SachinINDMI

s2 → SachinINDMI
 (Stack) (Heap A)

Note: \rightarrow String + "IND"

Addition if one operand is string
 then it causes Concatenation.

* Total 8 objects created and 2 are eligible
 for garbage collection.

s1

s1

s1 →

SCP

~~Sachin~~~~Sachin~~

IND

SCP

IND

MI

Q1.

String s1 = new String ("you cannot change");

String s2 = new String ("you cannot change");

System.out.println (s1 == s2); // false

String s3 = "you Cannot Change";

System.out.println (s1 == s3); // false

String s4 = "you Cannot Change";

System.out.println (s3 == s4); // true

String s5 = "you Cannot" + "Change";

System.out.println (s3 == s5); // true

final String s8 = "you Cannot";

String s9 = s8 + "Change";

System.out.print (s3 == s9);

// true

String s6 = "you Cannot";

String s7 = s6 + "Change";

System.out.println (s3 == s7); // false

System.out.print (s6 == s8);

// true

eg.4.

```
String s1 = new String("Sachin");
String s2 = s1.intern();
System.out.println(s1==s2);
```

Stack

 $s_1 \rightarrow$

Heap Area

Sachin

String s3 = "sachin";

System.out.println(s2==s3);

Scp

 $s_2 \rightarrow$ $s_3 \rightarrow$

Sachin

Interning: Using heap object reference, if we want to get corresponding Scp object, then we need to use intern() method.

eg.5

String s1 = new String("Sachin");

String s2 = s1.concat("tendulkar");

Stack

 $s_1 \rightarrow$

Heap Area

Sachin

String s3 = s2.intern();

String s4 = "Sachintendulkar";

System.out.println(s3==s4); // true

 $s_2 \rightarrow$

Sachin

tendulkar

Scp

Note:

Using heap object reference, if we want to get the corresponding Scp object and if object does not exists, their intern() will create a new object in Scp & return it.

 $s_3 \rightarrow$ $s_4 \rightarrow$

Sachin

tendulkar

Sachin

tendulkar

Importance of Scp:

1. In our programs if any String object is required to use repeatedly then it is not recommended to create multiple object with same content it reduces performance of the system and affects memory utilization.

- 2) We can create only one copy and we can reuse the same object for every requirement. This approach improves performance and memory utilization, we can achieve this by using "scp".
- 3) In SCP several references pointing to same object, the main disadvantage in this approach is by using one reference if we are performing any change the remaining references will be impacted. To overcome this problem immutability concept for String Objects were introduced.
- 4) According to this once we create a String Object we can't perform any changes in the existing object if we are trying to perform any change a new String object will be created hence immutability is the main disadvantage of SCP.

String Class Constructor (Commonly used)

String s = new String() → Create an empty String object

String s = new String(String Literals) → Create an object with String Literals on heap

String s = new String(StringBuffer sb) → Create an equivalent String object for string buffer.

String s = new String(char ch[]) → Create an equivalent String object for character array

String s = new String(byte [] b) → Create an equivalent String object for byte array

eg: char [] ch = {'j', 'a', 'v', 'a'};
 String s1 = new String(ch);
 System.out.println(s1);
 // java

byte [] b = {65, 66, 67, 68};
 String s2 = new String(b);
 System.out.println(s2);
 // ABCD

Important methods of String

1. `char charAt(int index)`
2. `String concat(String str)`
3. `boolean equals(Object o)`
4. `boolean equalsIgnoreCase(String s)`
5. `String subString(int begin)`
6. `String subString(int begin, int end)`
7. `int length()`
8. `String replace(char old, char new)`
9. `String toLowerCase()`
10. `String toUpperCase()`
11. `String trim()` — removes first and last blank space.
12. `int indexOf(char ch)`
13. `int lastIndexOf(char ch)`

eg: `String s = new String("Sachin");`

`System.out.print(s[3]);` //CE because in java string is

an object and accessing prohibited

`System.out.print(s.charAtIndex(3));` //h

`System.out.print(s.charAtIndex(1));` // String index out-of
bound exception.

Note:

Package `java.lang;`

Class String

```
{ public int length()
    {
        {
    }
}
```

Integer Array Class.

Class [I]

```
{ int length;
```

`s.length()`

(method of String class)

`array.length`.

property of
array class.

Code Snippets

1. `int n[7] = {{1,3},{2,4}};`
`for (int i = n.length-1; i >= 0; i--)`
`{ for (int y : n[i])` Ans: 2 4 1 3
`System.out.print(y); }`

2. `int nume1[7] = {1,2,3};`
`int nume2[7] = {1,2,3,4,5};`
`nume2 = nume1;` // Compiler for array assignment Compiler
`for (x : nume2)` will check only the type not length.
`System.out.print(x + " ");` Ans: 1 2 3

3. `int data[] = {2010, 2013, 2014, 2015, 2014};`
`int key = 2014; int count = 0;`
`for (int e : data)`
`{ if (e != key)` Ans: 0 found ✗
`{ continue; count++; }`
`}` Compile time error because
`System.out.print(count + " found");` "Count++" is unreachable code.

4. `int numbers[];`
`numbers = new int[2];`
`numbers[0] = 10; numbers[1] = 20;`
`numbers = new int[4];` // new object created
`numbers[2] = 30; numbers[3] = 40;` Ans: 0 0 30 40
`for (int x : numbers)`
`System.out.print(" " + x);`

5. `String days[] = {"sun", "mon", "wed", "sat"};` int wd = 0;
`for (String s : days)`
`{`

Switch (s)

```
{
    Case "sat": 
        Case "Sun": wd = 1;
        break;
    Case "mon": wd++;
    Case "wed": wd++ } } Ans : 3.
```

6. String [] str = {"A", "B"}
 int idx = 0;
 for (String s: str)
 { str[idx].concat("element" + idx);
 idx++; }
 for (idx = 0; idx < str.length(); idx++)
 { System.out.length(str[idx]); } Ans : AB.

Strings (Continued)

Q1. String s1 = "sachin"; s1, s3 → "Sachin" (SCP)
 String s2 = s1.toUpperCase(); || s2 → "SACHIN" (Heap Area)
 String s3 = s1.toLowerCase();

System.out.println(s1 == s2); || false

System.out.println(s1 == s3); || true.

Q2. String str = "";
 str.trim();
 System.out.println(str.equals("") + " " + str.isEmpty());
 || false false (because immutable)

Note: Whenever we print any reference, by default JVM will call "toString()" on the reference.

eg: Class Student

```
{ String name = "Sachin";
  int id = 10; }
```

In main {

```
Student std = new Student();
```

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

Out: Student@hexValue.

eg6 final StringBuffer sb = new StringBuffer("Sachin");
 sb.append(" tendulkar");

```
System.out.println(sb); // Sachin-tendulkar
```

Content of StringBuffer can be changed because
it is mutable even with final keyword.

sb = new StringBuffer("Kohli");

↑ sb if final can't be reused, sb can't point to new object,
because it's final

final vs immutability

- * final is a modifier applicable for variables, whereas immutability is only applicable for objects.
- * If reference variable is declared as final, it means we cannot perform re-assignment for the reference variable, it does not mean we cannot change/perform any change in that object.
- * By declaring the reference variable as final, we won't get immutability nature.
- * final and immutability are different concepts.

Note :

final variable ✓	immutable Variable ✗
final object ✗	immutable object ✓

- * String Builder, StringBuffer and all wrapper classes (Byte, Short, Long, Integer, Float, Double, Boolean, Character) are by default mutable.

Important methods of StringBuffer / String Builder

1. int length()
2. int capacity()
3. char charAt(int index)
4. void setCharAt(int index, char ch)
5. StringBuffer append(String s)
6. StringBuffer append(int i)
7. StringBuffer append (long l), [boolean, double, float]
8. StringBuffer append (int index, Object o)
(overloaded)
9. StringBuffer insert (int index, String s) [int, long, double, bool, float]
overloaded
10. StringBuffer delete (int begin, int end);

Constructors of StringBuffer

StringBuffer sb = new StringBuffer();

→ Create an empty StringBuffer object with default initial capacity of 16. Once StringBuffer reaches its maximum capacity a new StringBuffer object will be created
new capacity = (current capacity + 1) * 2;

Eg:

```
StringBuffer sb = new StringBuffer(19);
```

```
System.out.println(sb.length()); // 10
```

```
System.out.println(sb.capacity()); // 19
```

`StringBuffer sb = new StringBuffer("strings");`

It creates a String buffer object for the given string
with the capacity = s.length() + 16.

Code Snippets:

1. `String []arr = {"A", "B", "C", "D"};`
`for (int i=0; i<arr.length; i++)` Output: A Workdone
`{ System.out.print(arr[i] + " ");`
`if (arr[i].equals("C"))`
 `Continue;`
`System.out.println("workdone");`
`break;`

2. `String []str = new String[2];`
`int idz = 0;`
`for (String s: str)` Output: Null Pointer exception
`{ str[idz].Concat(" element." + idz);`
 `idz++; }`
`for (idz = 0; idz < str.length; idz++)`
`System.out.print(str[idz]);`

3. `StringBuffer sb = new StringBuffer("java");`
`String s = "java";`
`if (sb.toString().equals(s.toString()));` Output: Match!
`System.out.println("matched");`

```
else if (sb.equals(s))
    System.out("match 2");
```

```
else
```

```
    System.out("No-match");
```

4. `int[] a = new int[] ?` What is the array size?

Ans: Compile time error

5. `int[] a = new int[0]`

Ans: Code compiles fine.

6. `int[] a = new int[-5];` Size of the array?

Ans: Array index out of the bound exception is occurred

7. `long x = 42L, y = 44L;`

`System.out(" " + x + 2 + " ");` || 72 { return "foo"; }

`System.out(foo + x + 5 + " ");` || foo425 (because '+' with String is

`System.out(x + y + foo);` always concatenation)

result in addition and then concatenation $\rightarrow 86\text{foo}$.

String (continued)

* void setLength(int length)

→ It is used to consider only the specified no. of characters and remove all the remaining characters.

eg: `StringBuffer sb = new StringBuffer("sachinramesh");`

`sb.setLength(6);`

`System.out.println(sb);` || sachin

* void trimToSize()

→ this method is used to deallocate the extra allocated free memory such that capacity and size are equal.

eg:

StringBuffer sb = new StringBuffer(1000);

sb.capacity(); // 1000 sb.append("sachin"); // 1000 ~~is~~

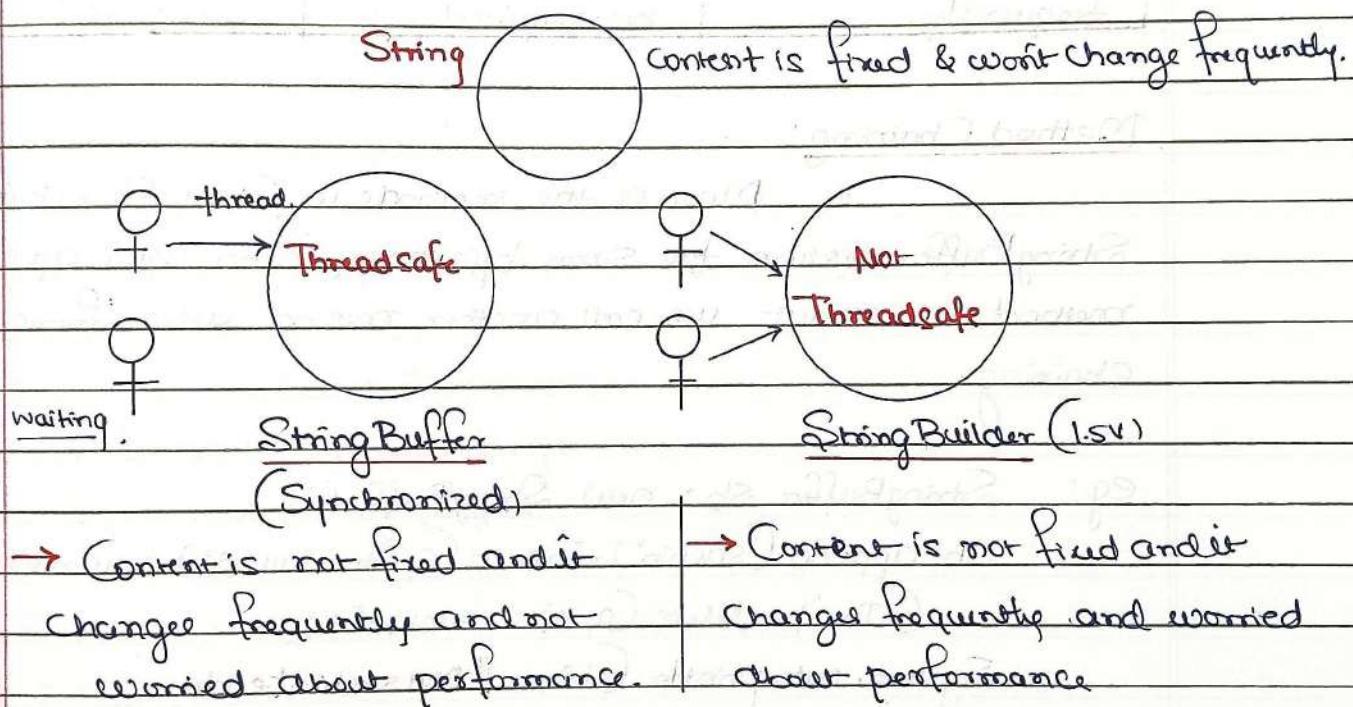
sb.trimToSize();

System.out.print(sb.capacity()); // 6

~~capacity~~,

* void ensureCapacity(int capacity)

→ it is used to increase the capacity dynamically based on our requirement



Every method present in StringBuffer is synchronized, so at a time only one thread are allowed to operate on StringBuffer object, it would create performance problem, to overcome this problem we should go for StringBuilder.

StringBuilder (1.5v): Same as StringBuffer (1.0v) with few differences.

StringBuilder:

- * No methods are Synchronized
- * At one time more than one thread can operate so it is not threadsafe.
- * Threads are not required to wait so performance is high.
- * Introduced in jdk 1.5 version

String	StringBuffer	StringBuilder
We opt if the content is fixed and it won't change frequently.	We opt if content changes frequently but thread safety is not required.	We opt if content changes frequently but threadsafety is not required.

Method Chaining:

Most of the methods in String, StringBuilder, StringBuffer return the same type only, hence after applying method on result we call another method which forms method chaining.

eg: `StringBuffer sb = new StringBuffer();
sb.append("Sachin").insert(6, "tendulkar").reverse().append("IND").delete(0, 4).reverse();
System.out.println(sb); // INDsachintendu`

Program: Copy one String to another.

```
String s1 = "iNeuron", s2 = "";  
for (int i=0; i < s1.length(); i++)  
{ s2 = s2 + charAt(i); }
```

`System.out.println(s2); //iNeuron`

Program : Uppercase a String without inbuilt methods.

```
String s1 = "iNeuron";
String s2 = "";
for (int i=0; i<s1.length(); i++)
{
    s2 = s2 + (char) (s1.charAt(i)+32);
}
System.out.println(s2);
```

Program : Reverse a String without inbuilt methods.

```
String s1 = "iNeuron";
String s2 = "";
for (int i = s1.length(); i>0; i--)
{
    s2 = s2 + s1.charAt(i);
}
System.out.println(s2);
```

Code Snippets:

1. String s = "SACHIN TENDULKAR";
int length = s.trim().length();
System.out.println(length); // 16

2. String s = "HelloWorld";
s.trim();
int i = s.indexOf(" ");
System.out.println(i); Ans: 5

3. String s1 = "Java";
String s2 = new String ("Java");
System.out.equals();

To print equal which code fragment should be added?

Ans : s1.equals(s2) { A }

if (s1.equals(s2))

- if (s1.equalsIgnoreCase(s2)) } B

else System.out ("not equal");

4. `System.out.println(" " == "");` - true

`Sysout.println(" ");` -

`System.out.println("A" == "A");` - true

`System.out.println("a" == "A");` - $a == A$ (because both in double quotes)

5. `String str = "Java Rocks!";`

`Sysout(str.length() + ":" + str.charAt(0));`

Ans: 11 : !

6. `String s1 = "OCA";`

`String s2 = "oCa";`

`Sysout(s1.equals(s2));` Ans: false

7. `String fname = "James", lname = "Gosling";`

`Sysout(fname = lname);`

Ans: "Gosling" because it is assigned (no E).

8. `String str = "Good";`

`change(str);`

`Sysout(str);`

Ans: Good.

public static void change(String s)

{ s.concat("-morning"); }

9. `StringBuilder sb = new StringBuilder("Good");`

`change(sb);`

`Sysout(sb);`

Ans: Good-morning

void change(String s)

{ s.append("-morning"); }

10. `String str1 = new String("Core");`

`String str2 = new String("CoRe");`

`Sysout(str1 == str2);`

Ans: Core.

11. Public class Test

```
{ public String toString()
{ return "TEST"; }
}
```

Public static void main (String [] args)

```
{ Test obj = new Test();
System.out (obj); }
```

}

Ans: TEST

12. String s1 = "OCAJP";

String s2 = "OCAJP";

System.out (s1 == s2); Ans: true

13. final String fname = "James";

String lname = "Gosling";

String name1 = fname + lname, name2 = fname + "Gosling";

String name3 = "James" + "Gosling";

System.out (name1 == name2); System.out (name2 == name3);

'false'

'true'

String Programming:

Program: Check if a string is a paliindrome or not.

String s1 = "NITIN", s2 = "";

for (int i = s1.length() - 1; i >= 0; i--)

```
{ s2 = s2 + s1.charAt(i); }
```

if (s1.equals(s2))

System.out.println("Yes");

else

System.out.println("No");

Program : Check for Anagram between two String

```

String s1 = "Race", s2 = "carE";
s1 = s1.toLowerCase();
s2 = s2.toLowerCase();
char [] ch1 = s1.toCharArray();
char [] ch2 = s2.toCharArray();

Arrays.sort(ch1); Arrays.sort(ch2);
if (Arrays.equals(ch1, ch2))
    System.out.println ("Its Anagram");
else
    System.out.println ("Its Not");

```

Program : Check for Pangram

```

String s1 = "The quick brown fox jumps over a the lazy dog";
s1 = s1.replace (" ", "");
char [] ch = s1.toCharArray();
int [] ar = new int [26];
boolean flag = false;

```

```

for (int i=0; i<ch.length-1; i++)
{
    int index = ch[i]-65;
    ar[index]++;
}

```

```

for (int i=0; i<ar.length; i++)
{
    if (ar[i] == 0)
    {
        System.out.println ("Not pangram");
        flag = true; break;
    }
}

```

```

if (flag == false) System.out.println ("Its pangram");

```

Encapsulation in Java:

- * Data hiding
- * Data binding
- * Providing Security
- * Providing Controlled access to data members.

Code. Class Student

{

```
private int age;           // instance variable / data members
private String name;      // private members can be only accessed
private String City;       inside the class.
```

```
void SetAge (int age)    // Setter
{ this.age = age; }
```

```
int getAge ()            // getter
{ return age; }
```

Note:

→ If a method is doing the activity of setting a value or data to its class variable (receiving data from outside) then it is called "Setter".

```
void SetName (String name)
{ this.name = name; }
```

→ Such a method whose sole purpose is to return its value to whoever calls the method, we called it as "getters".

```
void SetCity (String City)
{ this.City = City; }
```

```
String getCity ()
{ return City; }
```

{}

Public Class LaunchEncap

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

```
{ Student st = new Student();
```

```
st.age = 28; // Compiler error because age is private.
```

```
st.setAge(28); // valid
```

```
int age = st.getAge();
```

```
System.out.println (age); // 28
```

```
st.setName ("Hyder");
```

```
System.out.println (st.getName()); // Hyder }
```

```
}
```

Code Snippets :

```
1. int mask=0, count=0;
if ((5<7) || (7+count<10)) | mask++<10) mask = mask+1;
if ((6>8)^ false) mask = mask+10;
if (! (mask>1)&& 7+count>1) mask = mask+100;
System.out (mask + " " + count);
```

Ans : 2 + 0

```
2. int [ ] a = new int [3];
System.out.println (a); // [I@ ... { address
System.out.println (a[0]); // 0
```

3. `int[1][2] a = new int[3][2];`

`System.out.print(a); System.out(a[0]); System.out(a[0][0]);`
 ↪ [[I@... ↪ [I@... ↪ _0

4. `int[1][2] a = new int[2][1];`

`System.out(a); System.out(a[0]); System.out(a[0][0]);`
 ↪ [[I@ ↪ null ↪ Null Pointer Exception

5. `int[1][2] a = new int[3][2];` How many objects are created and

`a[0] = new int[3];` how many are eligible for garbage

`a[1] = new int[4];`

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

collection?

Object created: 11, GC-6

6. `int[1] a;`

`main()`

{ `Test t1 = new Test();`

`System.out(t1.a);` // Null

`System.out(t1.a[0]);` } // Null pointer exception.

7. `int[1] a = new int[3];` // declared at instance level

`System.out.print(Obj.a);` // null

`System.out(Obj.a[0]);` // array index out of bound exception.

8. `int[1] a;` // can't be used without initialized (local variable).

`System.out(a); System.out(a[0]);`

Ans: Compile time error

Note: A class in which all data members are private then
 that class is called "bean"

Encapsulation (Continued)

"this" Keyword: Within an instance method or a constructor, this is a reference to the current object - the object whose method or constructor is being called. You can refer to the any member of current object from within an instance method or a constructor by using this.

Class Student

```
{ private String name;
  private int age;
  private String city;
```

Public Student (String name, int age, String city) // Constructor

```
{ this.name = name;           // Constructor doesn't have return type.
  this.age = age;             // Constructor called when object created
  this.city = city; }         // name is same as class name.
```

```
public String getName()
{ return name; }
```

Note: * Constructor will be there by default, if we create one, Jvm will not create the default one.

```
public int getAge()
{ return age; }
```

* Constructors can have parameters or can be empty, and same arguments should be passed while creating object.

```
public String getCity()
{ return city; }
```

main()

```
{ Student1 std = new Student1 ("Rohit", 17,
  "Hyderabad"); }
```

// Object Created and initialized.

Constructors:

- * Constructors have same name as class name
- * Parameters of the Constructors are similar like method parameters.
- * Constructor is called when object is created or instantiated.
- * It will not have a return type and return statements.
- * Inside Constructors the first statement will be either Super() or this()

Super(): Calls → parent constructor } Constructor

this(): Calls → Constructor of same class } Chaining

- * We can write Super() / this() in first statement only.
- * Constructors can be overloaded → "Constructor overloading"

Purpose: If some statement has to be executed the moment we create an object → Constructor.

this() Method: Inside one constructor if we have a requirement to call another constructor we use "this()". If the constructor we are calling through this() is not available results in error.

"this"	"this()"
<ul style="list-style-type: none"> * It is a keyword * It refers to the current object 	<ul style="list-style-type: none"> It is a method It will call same class constructor.
method	Constructor
<ul style="list-style-type: none"> * Call explicitly by calling name * has return type and Statement 	<ul style="list-style-type: none"> it is called when object is created no explicit return type & Stmt.

Code Snippets.

1. `int [][] a = {{1,2,3}, {3,4}};`
`int [] b = (int []) a[1];`
`Object o1 = a;`
`int [] [] a2 = (int [] []) o1;`
`System.out (b[1]);` Ans: 4

2. `String [] x;`
`int [] a [] = {{1,2},{1,4}};`
`Object c = new long [4];`
`Object d = x;` || Compilation Succeed.

3. `int x = 5;`

```
main()
{ final Fizz f1 = new Fizz();
  Fizz f2 = new Fizz();
  Fizz f3 = FizzSwitch (f1,f2);
  System.out (f1==f3 + " " + (f1.x == f3.x));
  Output: true true.
```

Static Fizz FizzSwitch (f1,f2,
f2.x)

```
{ final Fizz z=x;
  z.x=6;
  return z; }
```

4. `int value = 0;`
`boolean setting = true;`
`String title = "Hello";`
`if (value || (setting && title == "Hello")) { return 1; }`
`if (value == 1 & title.equals ("Hello")) { return 2; }`

Class A a = new Class A(); Ans: Compilation error because
value is not boolean type

5. `int a=188, b=15, c=4;`
`System.out (2 * ((a*5) * (4+(b-3)/(c+2))));` Output: 36

6. `Sysout (23|2.0) || 11.5`
`Sysout (23*2.0) || 1.0`

7. `Sysout ("Hello" + 1 + 2 + 3 + 4); Output: Hello1234`

8. `Sysout (1+2+3+4 + "Hello"); Output: 10Hello`

9. `Sysout ("Output is :" + 10 != 5); Output: Compilation Error`

On String objects only '+' operator is allowed other operators would result in compile time error.

10. `Sysout ("Output is :" + (10 != 5)); Ans: Output is : true`

11. `int grade = 75;`

`if (Grade >= 60) Sysout ("Congrats");`

`Sysout ("You Passed");`

Output: Compilation error

`else`

`Sysout ("You Failed");`

because an interleaved string

before if and else.

12. `int grade = 60;`

`if (grade = 60)`

Output: Compilation Error because

`Sysout ("you passed");`

of assignment operator.

13. `String [] arr [] = { { ".f.", " ***" }, { "!!!", "@@@@", "####" } };`

`for (String str [] : arr)`

{ for (String s : str)

{ Sysout (s);

`if (s.length() == 4) break; }`

Output: *

`break; }`

Static Keyword:

Static Variable:

- * Static keyword is used before a variable
- * Memory is allocated during the class loading on heap area.
- * Memory is allocated once in heap
- * One copy of static variable used by all the objects and memory is not allocated all the time
- * Static variables can be called using class name
- * It is called as class variable because one copy is shared by all the objects.
- * They are object independent
- * They can be accessed inside static and non static elements.

Sequence of execution

1. Static Variables
2. Static block
3. Static method
4. Instance Variables
5. Non-static block
6. Constructors
7. methods (if called)

Execution flow of code (default)

1. Static variable → heap → during 'class loading'
2. Static block → to initialise → static variables (during class loading).
3. Static method → main method → other methods (if called)

Note : { 1. Static variables
2. Static blocks } executed by default in the sequence. Other methods will need to call.
3. main method }

When Object Created: `Demo d = new Demo();`

→ memory allocated for inst. variables
→ java block → constructor

eg: **Class Demo**

```
{ Static int a;
  Static int b;
```

Static

```
{ System.out.println ("Static block"); } || Static block
  a=10; b=20; }
```

Static void disp()

```
{ System.out.println ("Static method");
  System.out.println (a);
  System.out.println (b); } || Static method
```

int x, y;

|| instance variables

```
{ x=10; y=20;
  System.out.println ("Non Static Java Block"); }
```

Demo()

```
{ System.out.println ("Constructor"); } || Constructor
```

Void disp1()

```
{ System.out.println ("Non static method"); } || method
  System.out.println (x);
  System.out.println (y); }
```

Output:

Static block

Static method

10 20

Non Static block

Constructor

Non static method

10

20

main()

```
{ Demo.disp();
  Demo d= new Demo();
  d.disp(); d.disp1(); }
```

eg

Class Demo2{
 Static int a;
 Static
 { a=10; }
}

Static void disp()

{ System.out.println("Static Disp "+a); }

}

Public Class Launch{
 Static void disp2()
 { System.out.println("Disp 2"); }
}

public static void main (String [] args)

{ System.out.println("main method");
 disp2(); // directly calling (same class)

Demo2.disp(); // calling without object creation

Demo2 d = new Demo2();

d.disp(); } // calling after creating object.

}

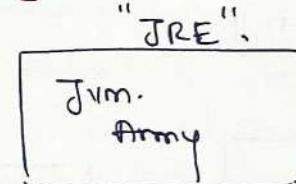
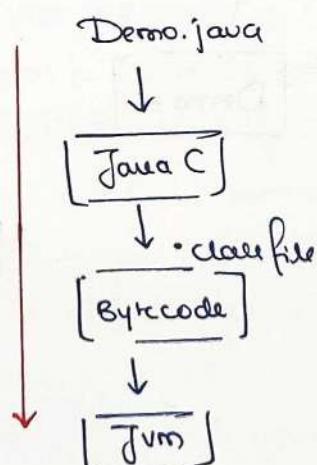
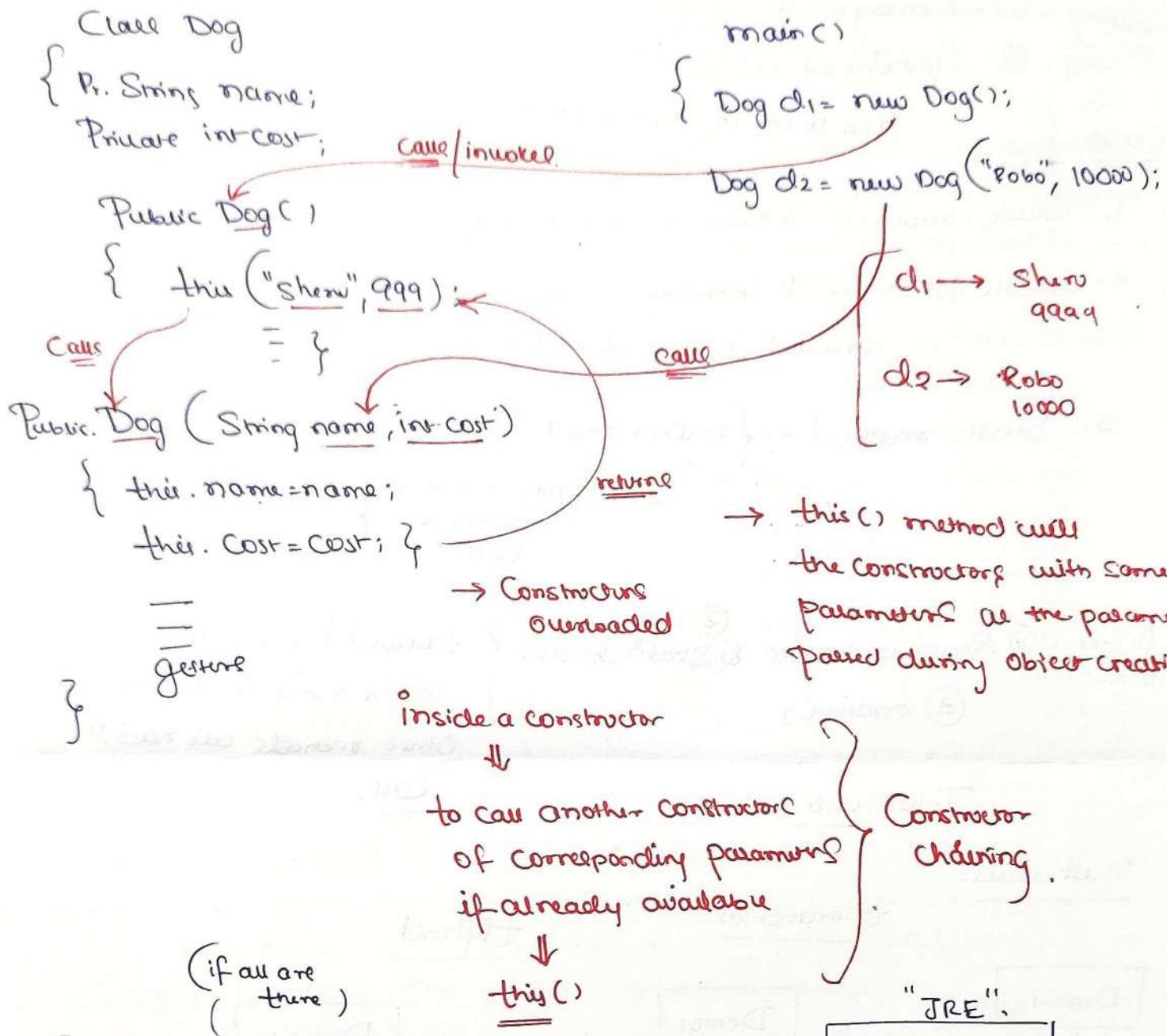
Output: main method

Disp 2

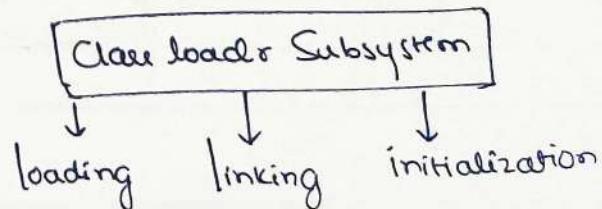
Static Disp 10

Static Disp 10

"this" method example:



- Jvm → ① class loader subsystem
- ② JVM if weapons
- ③ execution interpreter, JIT compiler ..



eg : Demo d = new Demo();

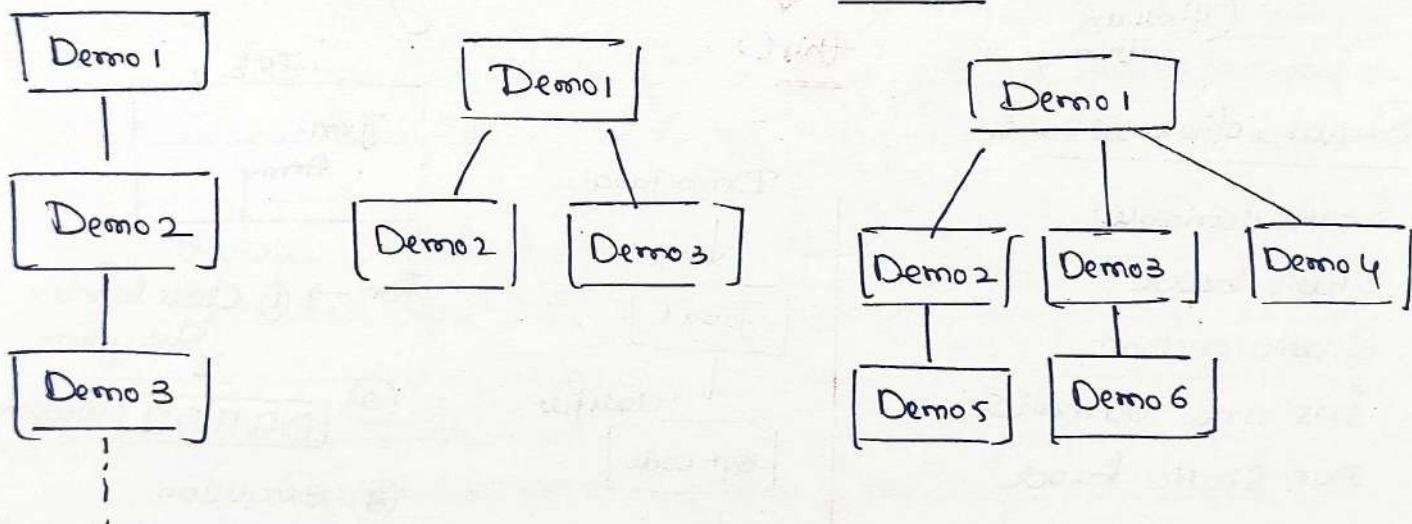
Execution { ① - memory for inst. variables
 =
 ② → java block → constructor
Execution flow: ↓ first to execute, then constructor

1. Static variables → Heap → class loading
 2. Static block → To initialize → Stat. variable?
executed during Class loading
 3. Static method — { main method (first)
= { other methods
= you should

Inheritance: Call

Multilevel.

Hierarchical, planar, hybrid



Static methodnon Static method

- * Static method can be called using class name (Generic)
- * Can be called using object reference also.
- * Object independent, need not have to create object.

- * Object creation is mandatory for using non static methods (Specific)
- * Can be called using object reference only
- * Object dependent, need to create object for calling method.

Simple interest Calculator Application:

```
import java.
```

```
class Farmer
```

```
{ private float pa, td, si;
  private float static ri;
```

```
static
```

```
{ ri = 2.5f; }
```

```
public void acceptInput()
```

```
{ Scanner Scan = new Scanner(System.in);
```

```
System.out.println("Dear, enter loan amount");
```

```
pa = Scan.nextFloat();
```

```
System.out.println("Dear, enter the time needed to repay");
```

```
td = Scan.nextFloat(); }
```

```
public void compute()
```

```
{ si = (pa * td * ri) / 100; }
```

```
public void disp()
```

```
{ System.out.println("Si is " + si); }
```

```
}
```

```

public class LaunchFarmer
{
    public static void main (String [] args)
    {
        Farmer f1 = new Farmer();
        Farmer f2 = new Farmer();

        f1.acceptInput(); f1.compute(); f1.disp();
        f2.acceptInput(); f2.compute(); f2.disp();
    }
}

```

Q1. When to use Static Variables?

Ans Whenever a common copy of data has to be shared among all the objects of a class and are not specific to any object then the data has to be used inside static

Code Snippets :

1. String fruit = new String (new char [] { 'm', 'a', 'n', 'g', 'o' });
switch (fruit)
{
 default: System.out ("Any fruit you do");
 Case "Apple": System.out ("Apple");
 Case "mango": System.out ("mango"); Output: Mango
 Case "Banana": System.out ("Banana"); Output: Banana
 break;
}

2. boolean flag = ! true;
System.out (! flag ? args [0] : args [1]); Output: Am.

3. int a = 3;
System.out (a++ == 3 || --a == 3 || a < 0 || a == 3); // true
System.out (a); // 4

4. `int a = 3;
m(++a, a++);
Sysout(a);` static void m (int i, int j)
 { i++; j--; }
 Output: 5.

5. `boolean flag = false;
Sysout ((flag = true) | (flag = false) || (flag = true));
Sysout (flag);` Output: true
 false

6. `boolean status = true;
Sysout (status = false || status = true | status = false);
Sysout (status);` Ans: Compilation error (because of assign operator).

7. `String msg = "Hello";
boolean [] flag = new boolean [7];
if (flag [6]) msg = "Welcome";
Sysout (msg);` Output: Hello.

8. `boolean flag1 = true, flag2 = false, flag3 = true; flag4 = false;
Sysout (!flag1 == flag2 != flag3 == !flag4); // false
Sysout (flag1 = flag2 != flag3 == !flag4); // true.`

9. `int score = 30; char grade = 'F';
if (50 <= score < 60) grade = 'D'; // Compilation error
- nesting of relational operator is not possible.`

Inheritance in Java

Code example

Class Demo1

```
{ String name = "Hyder";
  int age = 28;
```

```
void disp()
{
  System.out.println ("Demo1 "+age);
}
```

Class Demo2 extends Demo1

```
{ }
```

Public Class Launch

```
{ public static void main (String []args)
{
  Demo2 demo = new Demo2 ();
  demo.disp();
}
```

Output:

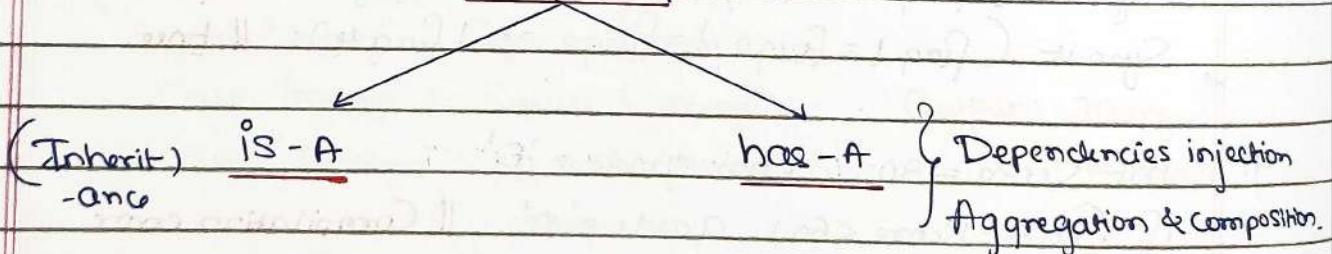
Demo1 28

(because of inheritance)

Demo2 is child/derived

Demo1 is parent/base
and it extends
"Object class".

Relationship



Also called:

- ① Parent-child relationship
- ② base class - Sub class
- ③ existing class - derived class.

Unrelated classes → We developed relation (code reusability)
→ through "extend" keyword.

Inheritance Key points:

- * Inheritance is a relationship.
- * Single inheritance is allowed (One class can extend another class).
- * Object class is parent of all class.
- * Multilevel inheritance is allowed.
- * One parent/base class can have any no. of child/subclass. (Hierarchical inheritance allowed).
- * Hybrid inheritance (Hierarchical + multilevel) allowed.
- * Multiple inheritance is not allowed (One class can have only one parent).
- * Cyclic inheritance is not allowed.
- * Private members will not participate in inheritance because to preserve the property of encapsulation.
- * Constructor will not participate in inheritance however parent class constructor will be called because of "super()" method call present inside the child class.

eg:

Class Parent

```
{
    private String name;
    Parent()
    {
        System.out.println("Parent Constructor");
    }
}
```

void disp()

```
{
    System.out.println("Parent");
}
```

}

Class Child

```
{
    // Child()
    {
        // { Super();
    }
}
```

void disp2()

```
{
    // name = hyder;
}
// private member
// can't access.
```

main()

```
{
    Child c = new Child();
    c.disp();
}
```

Output: Parent Constructor

Parent

eg

Class Parent

```
{ int a, b;
  Parent() { } // constructor
  { a=10;
    b=20;
    Sysout("Parent Const"); }
```

Parent(int a, int b) // Para

```
{ this.a=a; const.
  this.b=b;
  Sysout(" Parent para Const"); }
```

main()

```
{ Child ch = new Child();
  ch.disp(); }
```

Class Child extends Parent.

```
{ int x, y;
  Child() { } // Constructor
  { Super(); // adding this will call
    x=100; parent class. Const.
    y=200; }
```

Child(int x, int y) // param.

```
{ this.x=x;
  this.y=y; }
```

void disp()

```
{ Sysout(a); Sysout(b);
  Sysout(x); Sysout(y); }
```

Output: Parent Const

10 20 100 200

Another Variation:

Child (int x, int y)

```
{ Super(x, y); // adding parameters to Super will call the
  this.x=x; parametrized parent constructor
  this.y=y; } // atleast one constructor (in child class) must have
                Super method else not allowed.
```

main()

```
{ Child ch = new Child(1000, 2000);
  ch.disp(); }
```

Output: Parent para const

1000 2000 1000 2000

Access Specifiers : access specifiers help to restrict the scope of a class, constructor, variable, method, or data member.

There are four access specifiers:

1. public
2. protected
3. default
4. private (strongest)

(visibility)

eg
void display()
{
}

default

	Within a class	Outside class within package	outside package (is-A relationship)	outside package (no is-A relation)
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

eg

class Plane

code {
 public void takeOff()
 {
 System.out.println("Plane taking off");
 }
}

public void fly()
{
 System.out.println("Plane is flying");
}

public void landing()
{
 System.out.println("Plane is landing");
}

Class CargoPlane extends Plane

{
 public void fly()
 {
 System.out.println("Cargo plane flies");
 }
}

public void carryGoods()

{
 System.out.println("Carry goods");
}

Class PassengerPlane extends Plane

{
 public void fly()
 {
 System.out.println("Passenger plane flies");
 }
}

public void carryPassenger()

{
 System.out.println("Passenger plane
 carries passenger");
}

main (String [1 args])

```
{
    CargoPlane cp = new CargoPlane();
    PassengerPlane pp = new PassengerPlane();
    cp.takeOff(); // Inherited method
    cp.carryGoods(); // Specialised method
    cp.fly(); // Overridden method
    cp.landing(); // Inherited method
}
```

```

pp.takeOff();
pp.carryPassengers();
pp.fly();
pp.landing(); }
```

Output:

Plane is taking off.

Cargo plane carries goods
cargo plane flies

Plane is landing

Plane is taking off

Passenger plane carries passengers

Passenger plane flies
plane is landing

Rules to override methods:

1. We Cannot reduce the visibility of overridden method but we can increase it.
2. Return type of the overridden method must be same as that of overriding method in parent.
3. Return type of overridden method in child class can be different than that of parent class if it is co-variant return type (return type: is-A relationship)
4. Parameters of overridden method must be same as that of parent else it will be considered as specialized method
Considering method Overloading concept.

"Super ()" method:

- * Super method will be there inside constructor.
- * It will call the parent class.

Super Keyword: Super keyword is used in Child class to get the values of parent class.

eg: Class Demo1

```
{ int age = 28; }
```

Class Demo2 extends Demo1

```
{ int age = 27; }
```

void disp()

```
{ System.out.println(age); } // 27
```

System.out.println(super.age); } // 28 (parent class value)

}

}

Final Keyword: the final keyword is a non access modifier used for classes, variables and methods which makes them non changeable.

- * Final Variables acts as Constants we cannot change the value.
- * Classes declared as final don't participate in inheritance.
- * Methods declared as final can be inherited but cannot be overridden.

Code Snippets

1. Class Counter

```
{ int count;
private static void inc(Counter counter)
{ counter.count++; }
```

Counter C1 = new Counter();

Counter C2 = C1;

Counter C3 = null; C2.count = 1000;
inc(C2); }

Ans: No. of objects created: 1

2. String res = "";
 loop: for (int i=1; i<=5; i++)
 { switch (i)
 { Case 1: res += "UP"; break;
 Case 2: res += "TO"; break;
 Case 3: break;
 Case 4: res += "DATE"; break loop; }
 }

Sysout (res); // UP TO DATE

Sysout (String.join ("-", res.split (""))); // Up-To-DATE

3. String res = "";
 String [] arr = {"Dog", null, "friendly"};
 for (String s: arr)
 { res += String.join ("-", s); }
 Sysout (res); // Dog- null- friendly.

4. String [][] chs = new String [2] [];
 chs [0] = new String [2];
 chs [1] = new String [5]; int i=97;
 for (int a=0; a<chs.length; a++)
 { for (int b=0; b<chs[a].length; b++)
 { chs [a] [b] = "" + i; i++; }
 }
 for (String [] ca : chs) Output: 97 98
 { for (String c: ca) 99 100 null null null
 { Sysout (c + ""); }
 System.out.println(); }

5. String ta = "A";	ta = ta.concat (tb);	Sysout (ta);
ta = ta.concat ("B");	ta.replace ('c', 'd');	Out: A B C C
String tb = "C";	ta = ta.concat (tb);	

6. String Builder sb = new StringBuilder("B");
 sb.append(sb.append('A'));
 System.out.println(sb); Output: BABAB.

7. Class A

```
{ public String toString()
{ return null; }
}
```

main()

```
{ String text = null;      Output: (null null)
  text = text + new A();
  System.out.println(text); }
```

8.

8. Class SpecialString

```
{ String str;
  SpecialString(String str)
  { this.str = str; }
}
```

for (int i=1; i<=3; i++)

{ switch(i):

{ Case 1: arr[i] = new String("Java");
 break;

Case 2: arr[i] = new String("Java");
 break;

Case 3: arr[i] = new SpecialString("Java");
 break;

for (Object obj: arr)

{ System.out.println(obj); }

Output: null

Java

Java

Special String @ hash code

9. Class ToyStringClass extends String

{ String name; }

Output: Compile time error.

10. String name = "Sachin tendulkar".substring(4); || tendulkar

11. String s = "I".repeat(5);
 System.out.println(s); // IIIIII

12. `Sysout ("1".concat("2").repeat(5).charAt(7));` || 2.

13. To which of following class, you can create object `new`?

- a. String
- b. String Builder
- c. String Buffer

14. `String String = "String".replace('i', 'O');`
`Sysout (String.substring(2,5));` Output: rOn.

15. `Sysout ("Java" == new String("Java"));` || False

16. `if ("String".toUpperCase() == "STRING")`
`Sysout (true);`
`else` Output: false.
`Sysout (false);`

17. String, StringBuffer and String Builder - are final classes?
 Ans: Yes

18. `String str1 = "1", str2 = "2"; Str3 = "333";`
`Sysout (str1.concat(str2).concat(str3).repeat(3));`
 Output: 122333122333122333

19. `Sysout ("Java" + 1000 + 2000);` Output: Java10002000

20. `Sysout (1000 + 2000 + "Java");` Output: 3000Java

21. `Sysout (7.7 + 3.3 + "Java" + 3.3);` Output: 11.0 Java 3.3.

22. `String s1 = " "; Sysout (s1.isBlank());` || true (one blank space available)
`Sysout (s1.isEmpty());` || false

Class Parent

```
{ public static void disp()
    {
        System.out.println("Hello parent");
    }
}
```

Class Child {

```
public static void disp()
{
    System.out.println("Hello child");
}
```

- * Static methods participate in inheritance.
- * If you override static methods, it will be treated as specialized methods.

Ques.

```
main()
{
    Parent p = new Child();
    p.disp(); // Hello parent, because child class disp() is a specialized
              // function so you get parent class method
}
```

Child c1 = new Child();

```
c1.disp(); // Hello child, because it is specialized method
}
```

Polymorphism in Java

Class Parents

```
{ public void cry()
  { System.out ("Parents Crying"); }
```

assume
this is
not
there

```
/public void eat()
{ System.out ("Parents eating"); }
}
```

Class Child1 extends Parents

```
{ public void cry()
  { System.out ("Child1 Crying"); }
```

```
public void eat()
{ System.out ("Child1 eats less"); }
}
```

Output: Child1 crying

Child1 eats less.

Child2 crying

Child2 eats more

Class Child2 extends Parents

```
{ public void cry()
  { System.out ("Child2 Crying"); }
```

```
public void eat()
{ System.out ("Child2 eats more"); }
}
```

main() (valid)

```
{ Parents p = new Child1();
  p.cry(); // Child1 crying
```

// p.eat(); // directly using parent

type you cannot call Child class

Specialized methods

```
((Child1)p).eat(); // Child1 eating
```

// downcasting

```
Parents p2 = new Child2();

```

p2.cry(); // Child2 crying

```
((Child2)p2).eat();
```

// Child2 eats more.

Polymorphism:

Polymorphism in java is a concept that allows objects of different classes to be treated as common class. It enables objects to behave differently based on their specific class type.

- * Increases code reusability by allowing objects of different classes to be treated as objects of common class.
- * Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.
- * Improves readability and maintainability of the code.

Upcasting: Upcasting is the type casting of a child object to a parent object.

Downcasting: downcasting means the type casting of a parent object to a child object.

eg: Parent p = new Child();
 ((Child) p).eat();
 ('downcasting' 'calling specialized method')

eg2 Class Plane

```
{ public void takeOff()  

  { Sysout ("Plane taking off"); }
```

```
public void fly()  

{ Sysout ("Plane is flying"); }
```

```
public void landing()  

{ Sysout ("Plane Landing"); }
```

```
Class CargoPlane extends Plane  

{ public void fly()  

  { Sysout ("Cargo Plane flied"); }}
```

```
Class PassengerPlane ext. Plane  

{ public void fly()  

  { Sysout ("Passenger Plane  

  flied"); }}
```

Class FighterPlane extends Plane

```
{ public void fly()  

  { Sysout ("Fighter plane flied"); }}
```

Class Airport

```
{ public void permit //take  

  (Plane plane) plane  

  { plane.takeOff(); cargo and  

  plane.landing(); close all  

  plane.fly(); } work }
```

main()

```
{ CargoPlane cp = new CargoPlane();  

  FighterPlane fp = new FighterPlane();  

  PassengerPlane pp = new PassPlane();  

  Airport a = new Airport();  

  a.permit(cp);  

  a.permit(fp);  

  a.permit(pp); }
```

Abstraction in Java:

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstract Keyword:

- Abstract keyword cannot be applied to a variable
- Abstract keyword can be applied to a method and the method should not have a body.
- Abstract keyword is added to a class when there is atleast one abstract method.

eg

Abstract Class Loan

```
{ abstract public void dispInt();
  public void welcomeNote()
  { System.out.println("Welcome to xyz Bank"); }
```

Class HomeLoan extends Loan

```
{ public void dispInt()
  { System.out.println("RI is 12.1"); }
```

Class EducationLoan extends Loan

```
{ public void dispInt()
  { System.out.println("RI is 10.1"); }
```

main()

```
{ // Loan l = new Loan(); // we Cannot Create object of
  // Abstract class.
```

Loan l1 = new HomeLoan(); // we can create reference of abstract class.

l1.dispInt(); // RI is 12%.

l1.welcomeInt(); // Welcome to xyz Bank

Loan l2 = new EducationLoan();

l2.dispInt(); // RI is 10%.

l2.welcomeNote(); // Welcome to xyz Bank

}

Abstract Key points:

- * We Cannot Create Abstract Class object.
- * Abstract Class Can have all methods abstract. (100% abstraction).
- * Abstract Class can have both abstract and concrete methods.
- * Abstract Class can have all methods concrete.
- * The subclass / child class if extending abstract class then either have to implement abstract method (or) declare class abstract.
- * Constructor cannot be made abstract.
- * Abstract → incomplete Class
 - Child Class will implement } inheritance
- * We Cannot make Abstract Class as final, doing such is illegal.
- * We Cannot make Abstract method as final → illegal.
- * Variable Cannot be made abstract.
- * We can have Constructor in abstract class.

Area Calculator Program

```
import java.util.Scanner;
```

Abstract Class Shapes

{ float area;

Abstract public void input();

Abstract public void compute();

```
public void disp()
```

```
{ System.out.length ("The area is " + area);
```

```
}
```

Class Rectangle extends Shape

```
{ float length, breadth;
```

```
public void input()
```

```
{ Scanner Scan = new Scanner (System.in);
```

```
System.out.println ("Enter length of rectangle");
```

```
length = Scan.nextInt();
```

```
System.out.println ("Enter breadth of rectangle");
```

```
breadth = Scan.nextFloat(); }
```

```
public void Compute()
```

```
{ area = length * breadth; }
```

```
}
```

Class Square extends Shape

```
{ float length;
```

```
public void input()
```

```
{ Scanner Scan = new Scanner (System.in);
```

```
System.out.println ("Enter length of square");
```

```
length = Scan.nextFloat(); }
```

```
public void Compute()
```

```
{ area = length * length; }
```

```
}
```

Class Circle extends Shape

```
{ float radius;
```

```
final float pi = 3.14f;
```

```
public void input()
{
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter radius of circle");
    radius = scan.nextDouble();
}
```

```
public void Compute()
{
    area = pi * radius * radius;
}
```

Class Geometry

```
{ void Permit(Shapes s)
{
    s.input();
    s.compute();
    s.disp();
}}
```

public Class Launch Project

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

```

    Rectangle r = new Rectangle();
    Square ss = new Square();
    Circle c = new Circle();
}
```

```
Geometry g = new Geometry();
```

```

    g.Permit(r);
    g.Permit(ss);
    g.Permit(c);
}
```

Code Snippets

1. `for (int i=5; i>=1; i--)
{ System.out.print("*".repeat(i)); }`

**
*

2. `boolean opt = true;
switch(opt)
{ case true: System.out.println("True"); break;
default: System.out.println("False");
System.out.println("Done"); }`

What modification should be enabled to print "True Done"?
: Replace first line with
String s="True", case with
"True";

3. `StringBuilder sb = new StringBuilder(s);
String s = "";
if (sb.equals(s)) { System.out.println("match1"); }
else if (sb.toString().equals(s.toString())) { System.out.println("match2"); }
else System.out.println("match3"); Output: Match2`

4. `String text = "RISE";
text = text + (text = "ABOVE"); Output: RISE ABOVE
System.out.println(text);`

5. `StringBuilder sb = new StringBuilder("Java");
String s1 = sb.toString();
String s2 = "Java";
System.out.println(s1 == s2); Output: False`

6. `StringBuilder sb = new StringBuilder("Java");
String s1 = sb.toString();
String s2 = sb.toString();
System.out.println(s1 == s2); Output: False`

7. String str = "Java";
 String Builder sb = new String Builder ("Java");
 Sysout (str.equals (sb) + ":" + sb.equals (str));
 Output: false false

8. String Builder sb = new String Builder ();
 Sysout (sb.append (null).length()); Compilation Error

9. String Builder sb = new String Builder (s);
 sb.append ("0123456789");
 sb.delete (8, 1000); Sysout (sb); Output: 01234567

10. String Builder sb = new String Builder ("Hurray!");
 sb.delete (0, 1000); Sysout (sb); Output: o

Dependency Injection:

The process of injecting dependant object into target object is called as "Dependency Injection".

We can achieve dependency injection in 2 ways

- a) Constructor dependency injection
- b) Setter dependency injection

Constructor dependency injection: Injecting dependant object into target through a constructor.

Setter dependency injection: injecting dependent object into target object through a Setter.

eg Class Address || Dependency Object

{ }
Class Student || Target object
{ }
Address address; }
}
}

Relationships in Java

As part of Java application development, we have to use entities as per application requirements. In java application development, if we want to provide optimizations over memory utilization, code reusability, execution time, sharability then we have to define relationships between entities.

There are three types of relationships between entities

1. HAS-A relationship (extensively used in projects)
2. IS-A relationship
3. USE-A relationship

Difference between HAS-A and IS-A relationship:

Has-A relationship will define associations between entities in java applications, here associations between entities will improve communication between entities and data navigation between entities.

IS-A relationship is able to define inheritance between entity classes, it will improve "Code Reusability" in java applications.

Associations in Java

1. One to One Association (1:1)
2. One to many association (1:m)
3. Many to one association (m:1)
4. Many to many association (m:m)

To achieve associations between entities, we have to declare either single reference or array of reference variables of an entity class in another entity class.

eg

Class Address {

.....

}

Class Account {

.....

}

Class Employee {

.....

Address [] addr; // it will establish 1:m association

Account account; // 1:1 association

}

1. One to one association: it is a relationship between entities, where one instance of an entity should be mapped with exactly one instance of another entity.

eg: Every employee should have only one account

Account.java:

package in.inuron.bean;

// Dependent object

```
public class Account
{
    String accNo;
    String accName;
    String accType;
```

```
public Account (String accNo, String accName, String accType)
{
    super();
    this.accNo = accNo;
    this.accName = accName;
    this.accType = accType;
}
```

Employee.java:

Package: in.ineuron.bean;

|| Target Object

```
public class Employee
{
    private String eid;
    private String ename;
    private String eaddr;
```

Account account; || has-a relationship

public Employee (String eid, String ename, String eaddr,
Account account) || Constructor injection

```
{
    Super();
    this.eid = eid;
    this.ename = ename;
    this.eaddr = eaddr;
    this.account = account; }
```

```

public void getEmployeeDetails()
{
    System.out ("Employee details are:");
    System.out ("Empid : " + eid);
    System.out ("Empname : " + ename);
    System.out ("Emp address : " + eaddr);
    System.out.println();
    System.out ("Account details are");
    System.out ("Account no : " + account.accNo);
    System.out ("Account name : " + account.accName);
    System.out ("Account type : " + account.accType);
}

```

TestApp.java :

```

package in.inuron.main;
import in.inuron.bean.Account;
import in.inuron.bean.Employee;

```

```
public class TestApp {
```

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

```
        Account account = new Account ("ABC123", "Sachin", "Savings");
```

// Constructor injection

```
        Employee employee = new Employee ("Ino 10", "Sachin", "MI",
                                         account);
```

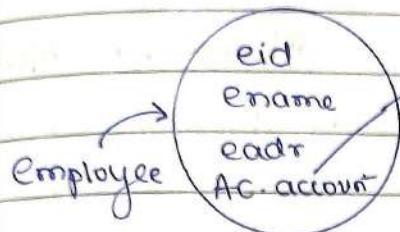
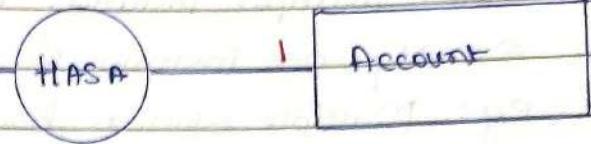
```
        employee.getEmployeeDetails(); }
```

```
}
```

Target Object

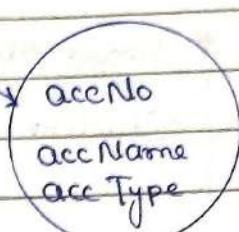


Dependent object



Dependency injection

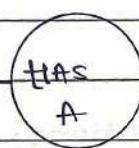
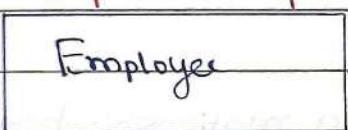
1. Constructor
2. getter and setter.



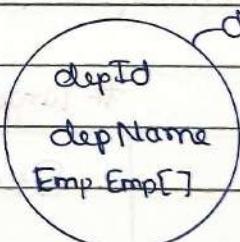
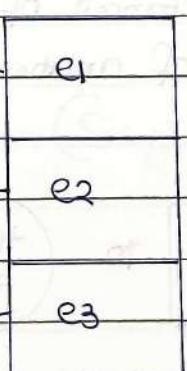
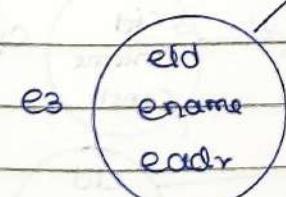
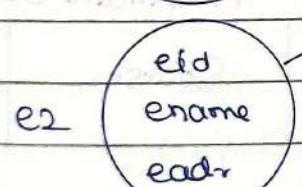
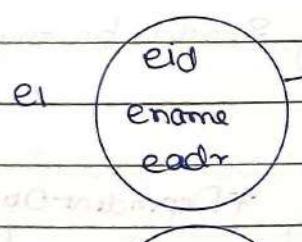
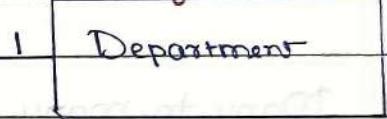
② One to many association: it is a relationship between entity classes, where one instance of an entity should be mapped with multiple instances of another entity.

eg: Single department has multiple employee.

Dependent Object



Target Object

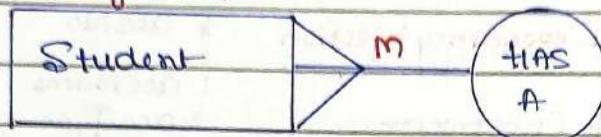


3.

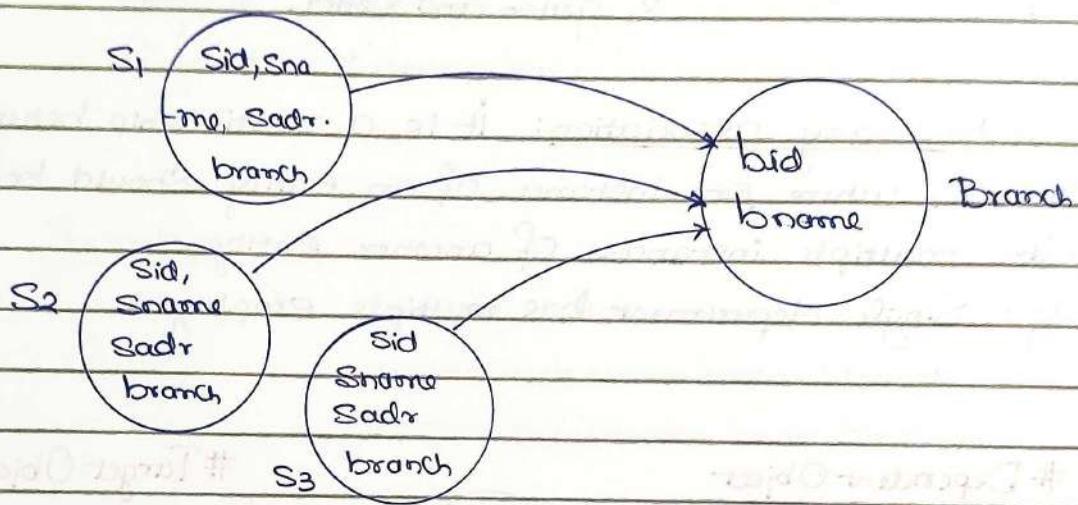
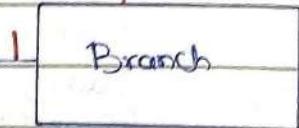
Many to One Association: It is a relationship between entities, where multiple instances of an entity should be mapped with exactly one instance of another entity.

e.g.: Multiple Students have joined with a single branch

Target Object



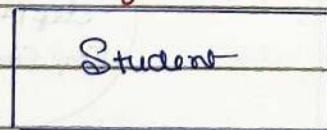
Dependent Object



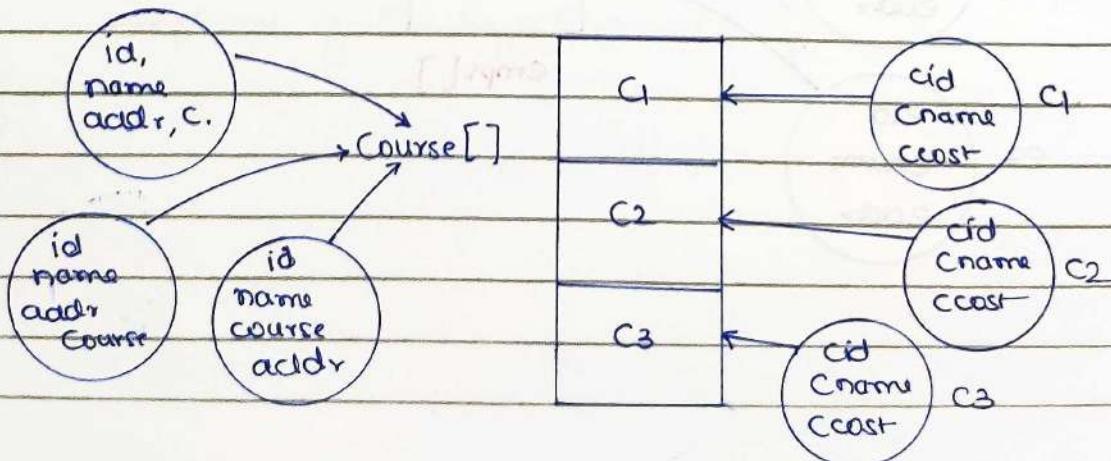
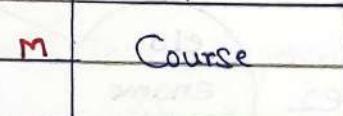
4.

Many to many associations: It is a relationship between entities, where multiple instances of an entity should be mapped with multiple instances of another entity.

Target Object



Dependent Object



Inner Class in Java

In java you can define a class within another class. Such a class is called nested class / inner class. Inner classes are divided into two categories : static and non-static.

Nested classes that are declared static nested are called static inner class and only inner class can be declared static.

Code: Class A

```
{ public void show()
    { System.out.println ("in Show"); }
```

Class B

```
{ public void Config()
    { System.out.println ("in Config"); }
```

|| inner class

|| object of inner class can be created in A.

public class FirstCode

```
{ public static void main (String[] args)
    {
        A obj = new A();
        obj.show(); || in Show
```

A.B obj1; || Creating object of inner class

obj1 = new A.B(); || Creating object of inner class (syntax only for static class)

obj1 = Obj.new BC(); || Creating inner class object from A obj
obj1.config(); } || in config

Code 2.Class Computer

```
{
    public void config()
    {
        System.out.println("In Computer config");
    }
}
```

public class SecondCode

```
{
    public static void main (String [] args)
    {
        Computer obj = new Computer();

        inner class {
            {
                public void config() // Overriding Computer method
                {
                    System.out.println ("Something new");
                }
            };
        }

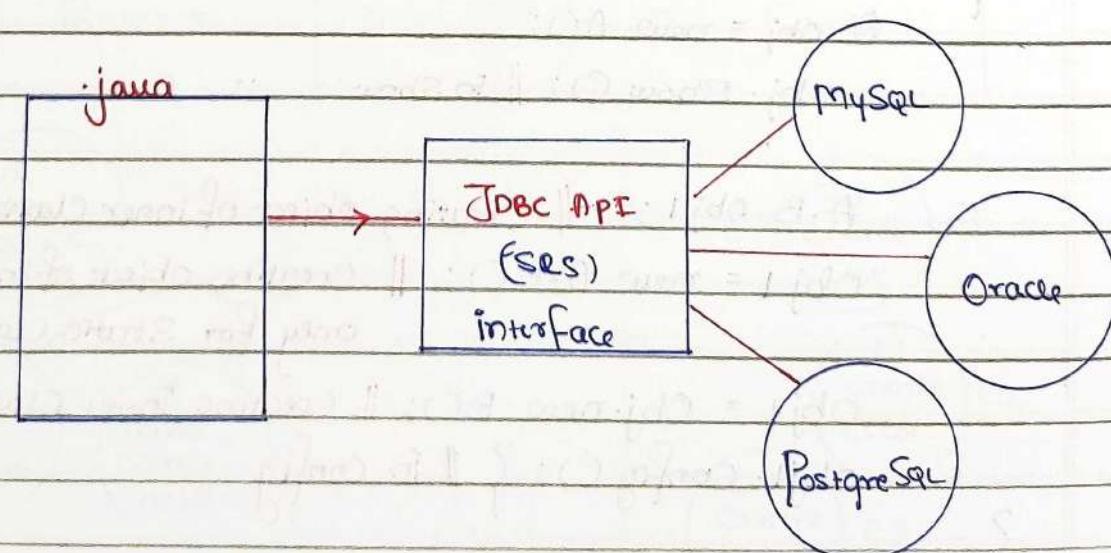
        obj.config(); // Something new.
    }
}
```

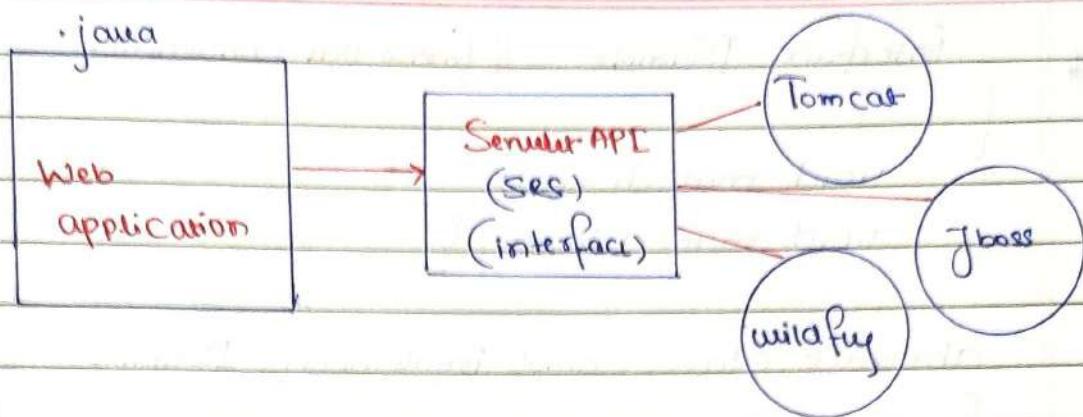
Note: This is an inner class, it's called anonymous class.

Note: There three classes are available and three .class files are generated.

Interface in Java

Def 1: Any service requirement specification (SRS) is called interface.



Note:

* Inside interface every method is always abstract whether we are declaring or not, hence interface is considered as 100% pure abstract class.

eg: interface Account

{

// 100% Abstract Class,

// methods are abstract and public by default

void withdraw();

void deposit(); }

Summary: Interface corresponds to Service requirement specification or Contract b/w Client and Service provider or 100% Abstract Class.

Declaration and Implementation of Interface.

- ① Whenever we are implementing an interface compulsorily for every method of that interface we should provide implementation. Otherwise we have to declare the class as Abstract Class in that case Child class is responsible to provide implementation for remaining methods.
- ② While implementing an interface method, it should be declared public. Otherwise results in Compile Time Error.

eg interface ISample // follow this convention

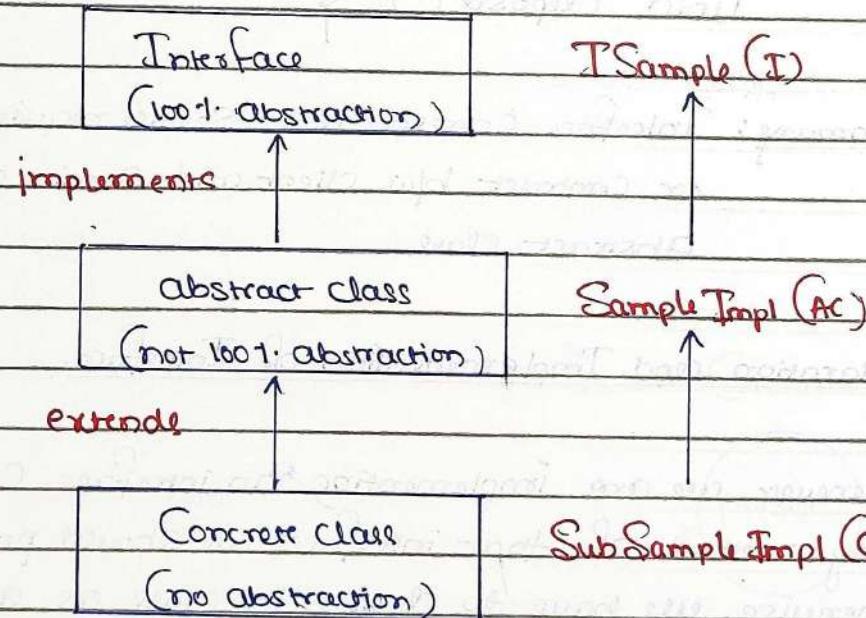
```
{
    void methodOne();
    void methodTwo();
}
```

abstract class Sample implements ISample

```
{
    public void methodOne() // method defined, but 2nd not defined
    { ... }
}
So class is abstract.
```

Class SubServiceProvider extends Sample

```
{
    @Override // indication to Compiler (methods are overridden)
    public void methodTwo() { ... } // method 2 defined
}
```



SampleImpl ref = new SubSampleImpl(); not good approach
 TSample ref = new SubSampleImpl(); good approach

Code Snippets

1. String Str = " ";

Sysout (Str.isBlank()); || true

Sysout (str.isEmpty()); || false

2 Which class of "String" have delete() and reverse() method?

Ans StringBuilder and StringBuffer.

3 java.lang.String class has append() method? Ans: No

4 String Str1 = "Java";

Sysout (Str2 == Str3); || false

String Str2 = Str1.intern();

Sysout (Str3 == Str1); || false

String Str3 = new String ("Java");

Sysout (Str1 == Str2) || true

5 String Str1 = "Java";

(Str1 == Str2); || true

String Str2 = Str1.intern();

Sysout (Str.equals(Str2)); || true

6 StringBuffer Sb1 = new StringBuffer ("1111");

StringBuffer Sb2 = Sb1.append (2222).append.reverse();

Sysout (Sb1 == Sb2); || false

7. StringBuilder and StringBuffer class has intern() method? No

8. Sysout (String.join (",", "1", "2", "3").concat (",").repeat (3).lastIndexOf (",",));

17

9. StringBuffer sb = new StringBuffer ("1111");

Sysout (sb.insert (3, false).insert (5, 33).insert (7, "one"));

Output: 111fa3. One31se1.

Interfaces (continued)

Difference between extends and implements

extends: One class can extend only one class at a time

eg: Class One { }

Class Two extends One { }

implements: One class can implement any number of interfaces at a time.

eg: Case 1

interface IOne

{ public void methodOne(); }

interface ITwo

{ public void methodTwo(); }

Class Demo implements IOne, ITwo

{ public void methodOne() { }; }

public void methodTwo() { }; }

Case 2: Class can extend as well as can implement an interface

1. reusability : → extends

2. interface : → implementation

Class Sample

{ public void m1() { } }

interface IDemo

{ void m2(); }

Class DemoImpl extends Sample implements IDemo

{ public void m2() { } }

Case 3 : An interface can extend any no. of interfaces at a time.

e.g.:

interface One
{ public void m1(); }

interface Two
{ public void m2(); }

interface Three extends One, Two
{ public void m1();
public void m2();
public void m3(); }

Q1. Which of the following is true?

- A class can extend any no. of class at a time.
- An interface can extend only one interface at a time.
- A class can implement only One " " " "
- A class can extend a class and implement an interface but not simultaneously.
- An interface can implements any no of interface at a time
- None of the above.

Q2. Consider the expression X extends Y which of the possibility of X and Y expression is true?

- Both X and Y should be classed
" " " " interface
- " " " Can be classes or interfaces
- No restriction

Interface Methods:

Every method present inside the interface is public and abstract.

Valid declarations are:

1. void m1();
2. public void m1();
3. abstract void m1();
4. public abstract void m1();

Why?

Public : → to make the method available for every implementation class.

Abstract : → implementation class is responsible for providing the implementation.

eg: JDBC API (java.sql.*)

Implementation given by : → MySQL, Oracle, Postgre

* Method allowed in interface is : public, abstract (2)

we can't use : static, private, protected, strictfp, synchronized, native, final.

Interface Variables:

* Inside the interface we can define variable.

* Inside the interface variables define requirement level constants.

* Every variable inside interface is "public static final" by default.

Why?

Public : → to make it avail. for implementation class object

Static : → to access it without using implementation class name

Final : → implementation class can access value without any modification.

- Note:
- * Since variable defined inside interface is public, static and final we cannot use modifiers like private, protected, transient, volatile.
 - * Since the variable is static and final, compulsorily it should be initialized at the time of compilation/declaration. Otherwise results in CE.

Interface Naming Conflicts

Case 1: if two interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

eg:

```
interface Left { public void m1(); }
interface Right { public void m1(); }
```

Class Test implements Left, Right {

 @Override

```
    public void m1() { ... }
```

Case 2: if two interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods act as overload methods.

eg:

```
interface Left { public void m1(); }
interface Right { public void m1(int i); }
```

Class Test implements Left, Right

{ @Override

```
    public void m1() { ... }
```

```
    public void m1(int i) { ... } }
```

Q1 Can a java class implements two interface Simultaneously?

Ans. Yes, possible but in both the interface method signature should be same, but not different return types

Case 3 : If two interface contain a method with same signature but different return types then it is not possible to implement both interface simultaneously.

eg:

```
interface Left{ public void m1(); }
```

```
interface Right{ public int m1(); }
```

Class Test implements Left, Right{

@ Override

```
public void m1(){...}
```

@ Override

```
public int m1(){...}
```

|| Invalid, So can't implement both simultaneously.

Q2 Can java class implements two interface simultaneously?

Ans Yes possible, except if two interface contain a method with same signature but different return types.

Variable Naming Conflicts :

Two interface can contains a variable with same name and there may be a chance of variable naming conflicts but we can resolve variable naming conflict by using interface names.

eg: interface Left{ int x=10; }

```
interface Right{ int x=20; }
```

Output: 10 20

public class Test implements Left, Right{

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

```
{ System.out (Left.x); System.out (Right.x); }
```

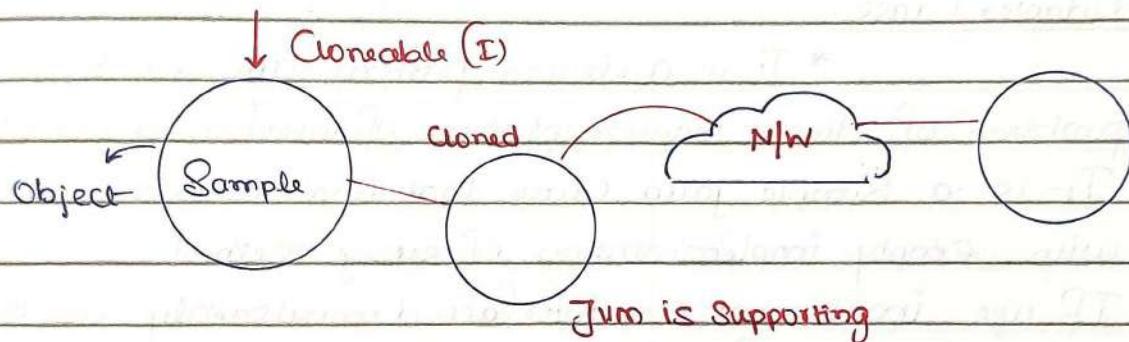
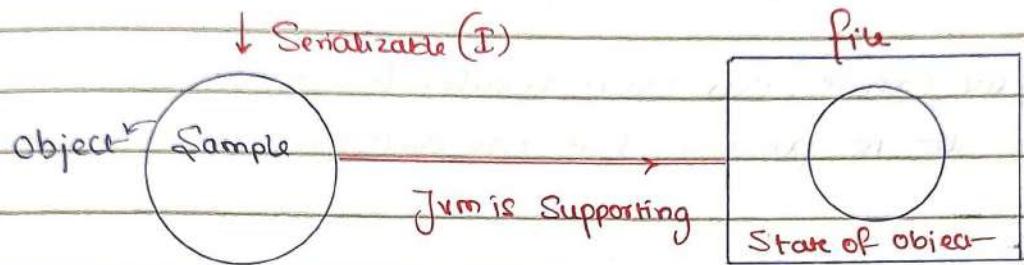
Note: We can also write an interface without any variable or abstract methods.

eg * `interface Serializable { }`

Class Sample implements Serializable { ... }

* `interface Cloneable { }`

Class Sample implements Cloneable { ... }



Marker Interface:

If an interface does not contain any method and by implementing that interface if our Object will get some ability such type of interface are called "Marker Interface" / "Tag Interface" / "Ability Interface".

eg:

1. Serializable: by implementing Serializable interface we can send that object across the network and we can save state of an object into the file

2. SingleThreadModel: by implementing it interface Servlet can process only one client request at a time so we can get "Thread Safety".

3. Cloneable: by implementing it our object is in a position to provide exactly duplicated clone object.

Q3 Without any method in marker interface how objects will get ability?

Ans Jvm is responsible to provide required ability.

Q4 Why Jvm is providing the required ability?

Ans To reduce the complexity of programming.

Q5 Can we create our own marker interface?

Ans Yes, it is possible but we need to customize Jvm.

Adapter Class:

- * It is a design pattern allowed to solve the problem of direct implementation of interface methods.
- * It is a Simple java Class that implements an interface only with empty implementation of every method.
- * If we implement an interface Compulsorily we should give the body for all the methods whether it is required or not. This approach increase length of code and reduces readability.

eg: interface X

```
{
    void m1();
    void m2();
    void m3();
}
```

Abstract class Sample Adapter X implements X

```
{
    public void m1() { } // giving dummy implementation
    public void m2() { }
```

```
public void m3() { }
```

Class Test extends Adapter

```
{ public void m3() { } } giving implementation only for required
{ System.out.print("I am from m3"); } method.
}
```

eg : Server (I)

↓ implements

Generic Server (Ac)

↓ extends

Http Server (Ac)

↓ extends

My Server (Class)

Interface

Abstract Class.

- * If we don't know anything about implementation and just have requirement specification, we should go for interface.

- * If we are talking about implementation but not completely then we should go for abstract class.

- * Every method present inside the interface is always public and abstract (default).

- * Every method present inside abstract class need not to be public and abstract.

- * We can declare methods as private, protected, final, static, synchronized, native, static.

- * No restriction on abstract class method modifiers.

- * Every variable : public, static, final.

- * Need not to be public static final.

* Can't declare Variable as private, protected, transient, volatile.	No restriction on access modifier.
* Every Variable Should be initialized at time of declaration.	No need to initialize Variables at time of declaration.
* We can't write static and instance block.	Static and instance block allowed.
* Can't write Constructor.	We can write Constructor.

Note:

Static block: - class file loading happens and used to initialize static variables.

instance block: during the creation of an object, just before the constructor call used for initialization of instance variables.

Constructor: during the creation of an object, used for initialization of instance variables.

Q6 What is the need of abstract class? Can we create an object class and does it contains constructor?

Ans Abstract class object cannot be created because it is abstract. But constructor is needed for constructor to initialize the object.

Q7 Why abstract class can contain constructor whereas interface cannot?

Abstract Class:

- * it is used to perform initialization of object
- * to provide value for the instance variable
- * to contain instance variable which are req for child object to perform initialization for those instance variable.

interface:

- * Every variable is always public, static and final so there is no chance of instance variable inside interface.
- So we should initialise at time of declaration
- * So constructor is not required

eg: abstract class Person

```
{
    String name;
    int age, height, weight;
```

Person (String name, int age, int height, int weight)

```
{
    this.name = name;
    this.age = age;
    this.height = height;
    this.weight = weight;
}
```

Class Student extends Person {

int rollNo;

int marks;

Student (String name, int age, int height, int weight, int rollNo, int marks)

```
{
    Super (name, age, height, weight, rollNo);
    this.rollNo = rollNo;
    this.marks = marks;
}
```

}

Q8. Can reference be created for abstract class?

Person p = new Student ("Sachin", 49, 5.6f, 71, 10, 100);

Q9. Can reference be created for interface?

1 Sample Sample = null;

Note: Every method present inside interface is abstract, but in abstract class also we take only abstract methods then what is need of interface concept?

Abstract Class Sample {

 public abstract void m1();

 public abstract void m2(); }

interface ISample {

 void m1();

 void m2(); }

→ We can replace interface concept with abstract class, but it is not a good programming practise.

eg: 1.

interface X

{ ... }

Class Test implements X

{ ... }

Test t = new Test();

eg: 2

abstract X

{ ... }

Class Test extends X

{ ... }

Test t = new Test();

* performance is high

* while implementing X we can extend one more class, through which we can bring reusability

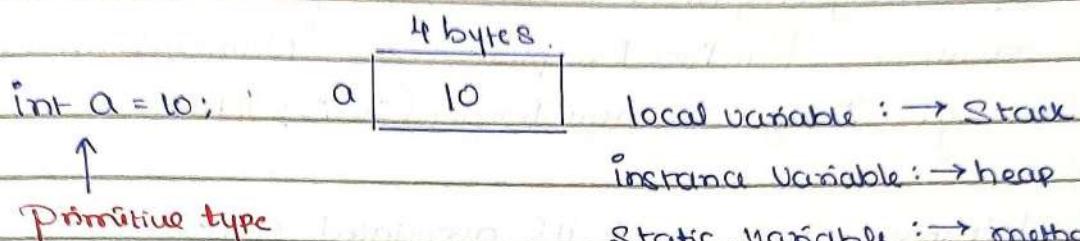
* performance is low

* while extending X we can't extend any other classes so reusability is not brought.

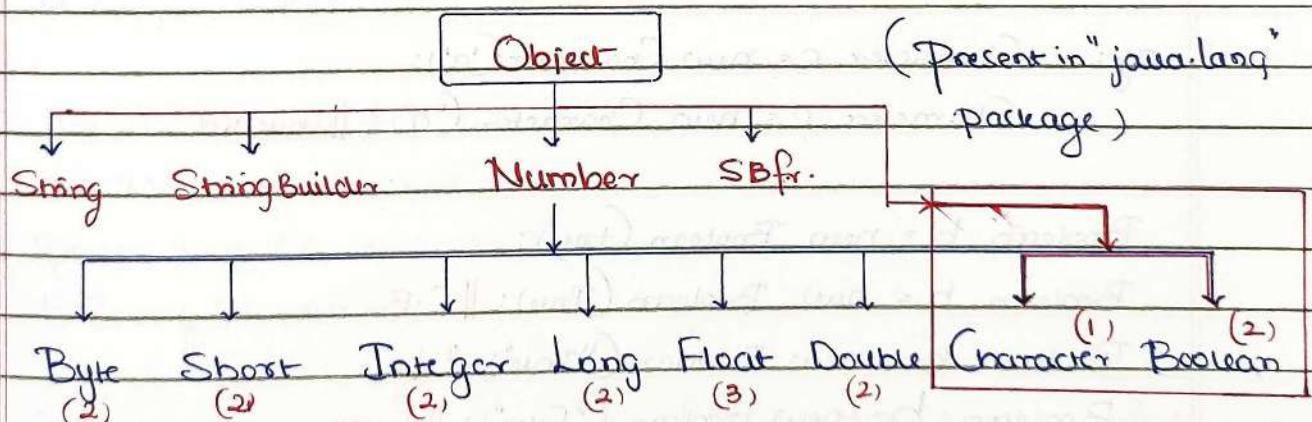
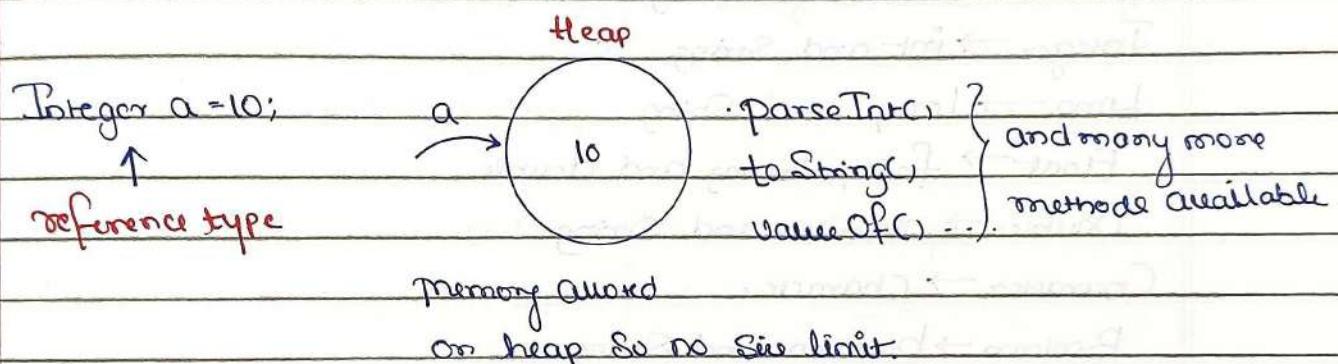
Note: if everything is abstract then it is recommended to go for interface.

Wrapper Class

Purpose : (i) To wrap primitive into object form so that we can handle primitive also just like object.
(ii) To define several utility functions which are required for the primitives.



JDK 1.5 v Wrapper Class are introduced



- * Object → `toString()` : returns the hashCode value of the object.
- * `toString()` : → Overridden to print the data present in the object.
- * `equals()` : → Overridden to compare the content present in the object.

```

cg: Integer i1 = new Integer(10);
      System.out.println(i1); // 10 (toString())
Integer i2 = new Integer("10");
      System.out.println(i2); // 10 (toString())
  
```

- * If String Argument is not properly defined then it would result in RunTime Exception called "NumberFormat-Exception".
- eg: Integer i = new Integer("tio"); // RE

Wrapper class and its associated constructor

Byte → byte and String

Short → short and String

Integer → int and String

Long → long and String

Float → float, String and double

Double → double and String

Character → Character

Boolean → boolean and String

eg: Character c = new Character('a');

Character c = new Character("a"); // invalid

Boolean b = new Boolean(true);

Boolean b = new Boolean(True); // C.E

Boolean b = new Boolean("True"); // true

Boolean b = new Boolean("False"); // false

Boolean b = new Boolean("nitin"); // false

- * If we are using Boolean, boolean value will be treated as true w.r.t insensitive part of "true", for all others it would be treated as "false".

- Note:
- * If we are passing String argument then case is not important and content is not important.
 - * If the content is Case Sensitive String of true then it is treated as true in all other cases it is treated as false.
 - * In Case of Wrapper Class, `toString()` is overridden to print the data.
 - * In Case of Wrapper Class, `equals()` is overridden to check content.
 - * Just like String Class Wrapper class are also treated as "Immutable Class".
- Immutable Class:
- If we create an object and if we try to make a change, with that change new object will be created and those changes will not reflect in old copy.

Q10 Can we make our undefined class as Immutable?

Ans Yes, by making the class final.

Wrapper Class Utility methods

1. `valueOf()` method.
2. `XXXValue()` method.
3. `ParseXXX()` method
4. `toString()` method.

(i) `public static wrapper valueOf (String data, int radix);` } throw
 (ii) `public static wrapper valueOf (String data);` } number
 (iii) `public static wrapper valueOf (int data);` } format
 exception.

1 `valueOf()` method:

- * To create a wrapper object from primitive type or

String we use `valueOf()`.

- * It is alternative to constructor of wrapper class, not suggested to use.
- * Every wrapper class, except character class contain static `valueOf()` to create a wrapper object.

(i)

Eg: `Integer i1 = Integer.valueOf("1111");`

`Sysout(i1); //1111`

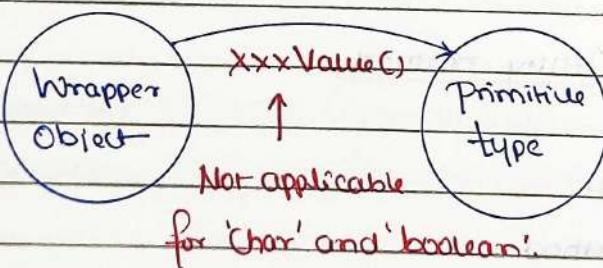
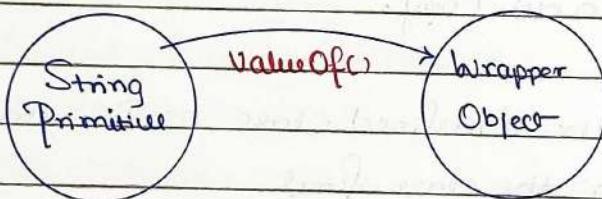
`Integer i2 = Integer.valueOf("1111", 2);`

`Sysout(i2); //15`

`Integer i3 = Integer.valueOf("ten"); //RE: NumberFormatException`

Eg (ii) `Integer i1 = Integer.valueOf(10);`

`Double d1 = Double.valueOf(10.5);`



2. xxxValue()

We can use `xxxValue()` to get primitive type for the given wrapper object. These methods are part of every Number type Object. All these classes have 6 methods.

Eg:

`Integer i = new Integer(130);`

`Sysout(i.byteValue()); // -126`

`Sysout(i.shortValue()); // 130`

`Sysout (i.intValue()); //130`

`Sysout (i.longValue()); //130`

`Sysout (i.floatValue()); //130.0`

`Sysout (i.doubleValue()); //130.0`

3. CharValue()

Character class contains CharValue() to get Char primitive for the given Character object.

eg: Character c = new Character ('c');

Char ch = c.charValue();

4. Boolean Value()

Boolean class contains booleanValue() to get boolean primitive for the given Boolean object.

eg: Boolean b = new Boolean ("initial");

boolean b1 = b.booleanValue();

Sysout (b1); // false

In total xxxValue() are 36 in number

xxxValue() → Convert Wrapper Object → Primitive

5. Parse xxx()

We use parsexxx() to convert String object into primitive type.

eg: int i = Integer.parseInt("10");

double d1 = Double.parseDouble ("10.0");

boolean b = Boolean.parseBoolean ("true");

* Every wrapper class, except Character class has parsexxx() to convert String into primitive type.

eg 1.2. WAP to take inputs from Command line and perform arithmetic operations.

```
int i1 = Integer.parseInt(args[0]);
int i2 = Integer.parseInt(args[1]);
System.out.println((i1 * i2) + (i1 - i2));
```

form-2: public static primitive parseXXX (String, int radix)
Radix → range from 2 to 36

* Every integral type wrapper class (Byte, Short, Int, Long)
Contains the following parseXXX() to convert specified radix
String to primitive type

eg: int i = Integer.parseInt("111", 2); //15.

6. toString()

: To convert wrapper object or primitive to String.
Every wrapper class contains toString().

Form 1: public String toString()

1. Every wrapper class (including character) contains above method to convert wrapper object to String.
 2. It is the overriding version of the Object class toString() method.
 3. Whenever we are trying to print wrapper object reference internally this "toString()" method only executed.
- eg: Integer i = Integer.valueOf("10");

Form 2: public static String toString(primitive type)

Every wrapper class contains a static toString() method to convert primitive to String.

eg: String s = Integer.toString(10);

String s = Character.toString('a');

String s = Boolean.toString(true);

form 3: Public static String toString (primitive p, int radix)

Integer and long class contain the following static toString() method to convert the primitive to specified radix string form.

eg: String s = Integer.toString(15, 2);
System.out(s); // 1111

form 4:

Integer and Long class contain the following toXXXString() methods:

Public static String toBinaryString(primitive p);

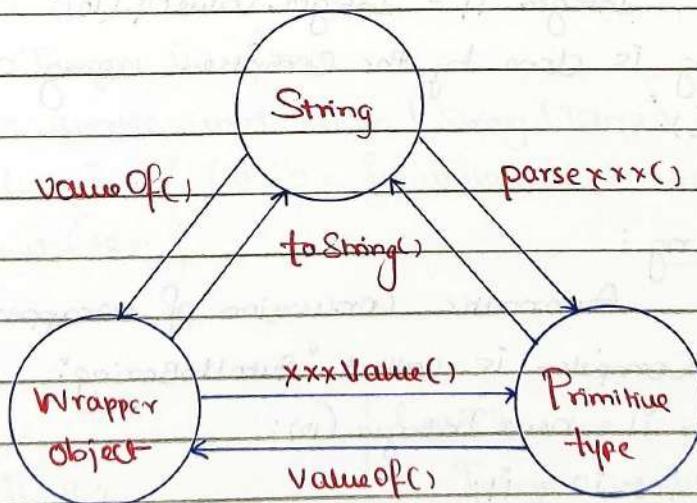
Public static String toOctalString(primitive p);

Public static String toHexString(primitive p);

eg: String s1 = Integer.toBinaryString(7); // 111

String s2 = Integer.toOctalString(10); // 12

String s3 = Integer.toHexString(10); // a



Note: String data = String.valueOf('a'); // static factory method

String data = "Sachin".toUpperCase(); // instance factory method,

AutoBoxing and Auto Unboxing

Until 1.4 version, we can't provide wrapper class objects in place of primitive and primitive in place of wrapper object, all the required conversions should be done by the program. But from Jdk 1.5 version onwards, we can provide primitive in place of wrapper and in place of wrapper we can keep primitives also.

All the required conversions will be done by the compiler automatically, this mechanism is called "AutoBoxing" and "AutoUnboxing".

eg: `ArrayList al = new ArrayList();
al.add(10);`

AutoBoxing:

Automatic conversion of primitive type to wrapper object by the compiler is called "AutoBoxing".

`Integer ii = 10;`

↓ After compilation the code would be

`Integer ii = Integer.valueOf(10);`

AutoBoxing is done by the compiler using a method called "valueOf()".

AutoUnboxing:

Automatic conversion of wrapper object to primitive type by Compiler is called "AutoUnBoxing".

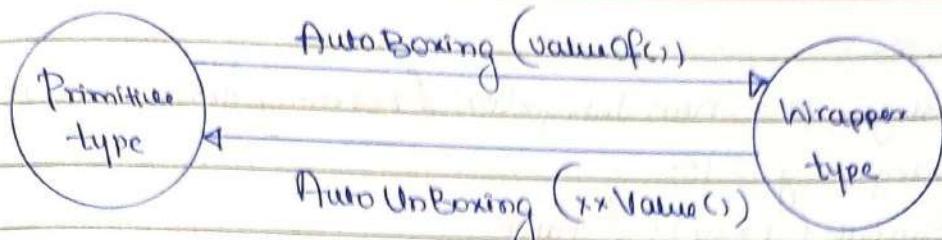
`Integer ii = new Integer(10);`

`int i2 = ii;`

↓ Compiler converts Integer to int

`int i2 = ii.intValue();`

AutoUnBoxing is done by compiler using a method called "xxxValue()";



Compiler will automatically do the conversions from Jav 1.5 version

eg Class Test

```
{ static Integer i1 = 10; // AutoBoxing
```

```
public static void main (String [] args)
{
    int i2 = i1; // AutoUnBoxing
    m1(i2); }
```

```
public static void m1 (Integer i2) // AutoBoxing
{
    int k = i2; // AutoUnBoxing
    System.out.println (k); } // 10
}
```

eg 2 Class Test

```
{ static Integer ii; // null
public static void main (String [] args)
{
    int i2 = ii; // int i2 = ii.intValue() :: Null Pointer Exception
    System.out.println (i2); }
}
```

Case 3:

```
Integer i1 = 10;
Integer i2 = i1; i1++;
System.out.println (i1 + " " + i2); // 10
System.out.println (i1 == i2); // false
```

Case 4:

```
Integer x = new Integer (10);
Integer y = new Integer (10);
System.out.println (x == y); // false
```

Case 5:

Integer x = new Integer(10); // memory from heap area

Integer y = 10;

System.out(x==y); // false

Case 6:

Integer x = new Integer(10);

Integer y = x; // reference is used So pointing to same object

System.out(x==y); // true

Case 7:

Integer x = 10;

Integer y = 10;

System.out(x==y); // true

Integer i = 1000;

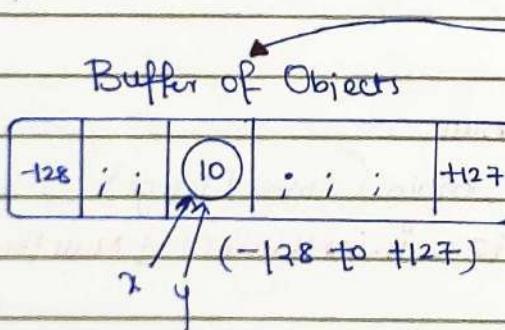
Integer j = 1000;

System.out(i==j); // false

* Compiler uses "valueOf()" for AutoBoxing



Implemented in intelligent way in wrapper class



At the time of loading the class

JVM buffer will already contain the objects which are to be used during AutoBoxing range (-128 to 127)

Note:

1. To implement auto boxing concept in wrapper class a buffer of object will be created at the time of class loading.
2. During auto boxing if an object has to be created first JVM will check whether the object is already available inside buffer or not.
3. If available JVM will reuse buffer object instead of creating one.

4. If the object is not available inside buffer, Jvm will create a new object in the heap area, this approach improves the performance and memory utilization.

But this buffer concept is applicable for few cases

1. Byte, Short, Integer, Long : -128 to +127

2. Character : 0 to 127

3. Boolean : true, false

In remaining Cases new object will be created

eg : Integer a=128;

Integer y=128;

Systout (x==y); // false

Integer a=127;

Integer b=127;

Systout (a==b); // true

Boolean b1=true;

Boolean b2=true;

Systout (b1==b2); // true

Double d1 = 10.0;

Double d2 = 10.0;

Systout (d1==d2); // false

Code Snippets

1. final int i1=1;

final Integer i2=1; // memory will be decided at runtime because it is

final String s1=":ONE";

wrapper class object.

String Str1 = i1+s1, Str2 = i2+s1;

Systout (Str1 == "1:ONE"); // true

Systout (Str2 == "1:ONE"); // false

2. String javaworld = "JavaWorld", java="Java", world="World"; // Sc

java=world; // Jvm → JavaWorld (HeapArea)

Systout (java == javaworld); // false

3. `StringBuilder sb = new StringBuilder("SpaceStation");
sb.delete(5,6).insert(5,'$').toString().toUpperCase();
System.out.println(sb); // Space Station`

4. `String s1 = "OCP", s2 = "ocp";
System.out.println(s1.equalsIgnoreCase(s2)); What should be added to print true?
Ans: s1.equalsIgnoreCase(s2), s1.length == s2.length()`

5. `String str = "PANIC"; StringBuilder sb = new StringBuilder("THE").
System.out.println(str.replace("N", sb)); // PATHETIC`

6. `boolean f1 = "Java" == "Java".replace("J", "J"); // true
boolean f2 = "Java" == "Java".replace("J", "J"); // true
System.out.println(f1 & f2); // true.`

7. `StringBuilder sb = new StringBuilder("TOMATO");
System.out.println(sb.reverse().replace("O", "A")); // Compilation error`

8. `String[] strings = "iNeuron-Techology-Private-known-for-Java".
split("-", 3);
for (String string : strings) System.out.println(string);`
Output: iNeuron
Technology
Private-known-for-Java
Specified index for an array.

Output: iNeuron

Technology

Private-known-for-Java

{ Because 3 indices was

specified in split.

9. `String s1 = "null"+null+1; System.out.println(s1); // nullnull1`

10. `System.out.println(1+null+"null"); // Compile time error`

11. `String[] strings = {"Java", "Hibernate", "Spring"};
String l = String.join("-", strings); // Java-Hibernate-Spring`

Var arg method

It Stands for Variable argument methods. In Java if we have variable number of arguments, then compulsory new method has to be written till Jdk 1.4 version.

But in Jdk 1.5 version, we can write single method which can handle variable no. of arguments but all of them should be of same type.

Syntax: methodOne (dataType ... variable name)

... → It stands for "ellipse".

eg: public void int add (int ... x)

(...) JVM internally uses array representation to hold values of x

Object.add(); → new int[] {}

Object.add(10); → new int[] {10}

Object.add(10, 20); → new int[] {10, 20}

eg) Class Demo

```
{ public void add (int ... x)
  { int total = 0;
    for (int i=0; i<x.length; i++) total += x[i];
    System.out.print (total);
  }
}
```

public static void main (String [] args)

```
{ Demo d = new Demo();
```

d.add(); // 0

d.add(10); // 10

d.add(10, 20, 30); // 60 ?

}

Valid Signatures

1. void add (int ... x)
2. void add (int ... x)
3. void add (int ... x)

Case 2 : We can mix normal argument with var argument.

public void methodOne (int x, int... y) ✓

public void methodOne (String s, int... x) ✓

void methodOne (int... x, String s) // Invalid

Case 3 : While mixing Var arg with normal argument var arg should always be last.

public void add (int... a, String b) Invalid.

Case 4 : In an argument list there should be only one var argument.

public void add (int... x, int... y) Invalid

Case 5 : We can overload var arg method, but var arg method will get a call only if none of matches are found (just like "default" of switch case).

eg: Class Test

```
{ public void m1 (int... x)
  { System.out ("var arg method"); }
```

public void m1 (int x)

```
{ System.out ("int method"); }
```

public static void main (String [] args)

```
{ Test t = new Test ();
```

t.m1(); // var arg method

t.m1 (10, 20); // var arg method

t.m1 (10); } // int method

```
}
```

Case 6: public void m1(int... x) → it can't be replaced as int[1x]
but vice versa.

Case 7:

public void m1(int... x)

public void m1(int[1x]) //CE.

We cannot have two methods with same signature

Single dimension array v/s Var Arg method

1. Whenever Single dimension array is present we can replace it with Var Arg.

eg: public static void main(String[1 args]) → String... args.

2. Whenever Var Arg is present we cannot replace it with Single dimension array.

eg: public void m1(String... args) → String[1args] (invalid)

Note:

m1(int... x) → We can call to this method by passing group of int values and it will become 1D array.

m1(int[1x]) → We can call to this method by passing 1D array only

eg: 1. public void m1(int... x) { ... }

Test t = new Test();

t.m1(10, 20, 30); // Treated as 1D array

2. public void m1(int[1...2])

int[1] a = {10, 20, 30}; int[1] b = {40, 50};

Test t = new Test();

t.m1(a, b); // Treated as 2D array

Wrapper Class

1. AutoBoxing
2. Widening (implicit type casting done by compiler for both primitive and wrapper class).
3. Var-args.

Case 1: Widening v/s AutoBoxing

Class Test

```
{
    public static void m1 (long l)
    {
        System.out ("widening");
    }
}
```

```
public static void m1 (Integer i)
{
    System.out ("autoboxing");
}
```

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

```
{
    int x = 10;
    m1 (10);
}
```

// int → implicit type casting → long

Output: widening

Case 2: Widening v/s Var-arg method

```
public static void m1 (long l)
```

```
{ System.out ("widening");
}
```

```
public static void m1 (int ... i)
```

```
{ System.out ("var-arg method");
}
```

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

```
{
    int x = 10;
    methodm1 (10);
}
```

// int → typecasting → long

Output: widening

Case 3: Autoboxing v/s var arg method

```

public static void m1(Integer i)
{
    System.out("autoboxing");
}

public static void m1(int... i)
{
    System.out("var-arg method");
}

public static void main(String[] args)
{
    int x=10; m1(x); } // int → type casting → long, float, double
// int → autoboxing → Integer

```

Output: autoboxing.

Case 4:

```

public static void m1(Long l)
{
    System.out("long");
}

public static void main(String[] args)
{
    int x=10; m1(x); } // CE: Can't find the method.

```

- * Widening followed by autoboxing is not allowed in Java, but autoboxing followed by widening is allowed.

Case 5:

```

public static void m1(Object o)
{
    System.out("Object");
}

public static void main(String[] args)
{
    int x=10; m1(x); } // Autoboxing → int → Integer
// Widening → Integer → Number, Object

```

Output: Object

Valid declarations and invalid declarations:

1. int i=10; ✓

2. Integer I = 10; ✓ Autoboxing (ValueOf())
3. int i = 10L; ✗ long → int
4. long l = 10L; ✓ Autoboxing
5. long l = 10; ✗ Autoboxing → Integer, Number so invalid.
6. long l = 10; ✓ int → long
7. Object o = 10; ✓ Autoboxing → Integer → Object
8. double d = 10; ✓ int → Double
9. Double d = 10; ✗ Autoboxing → Integer → Number, Object
10. Number n = 10; ✓ Autoboxing → Integer → Number

New v/s newInstance

1. new is an operator to Create objects, if we know class name at the beginning then we can create an object by using new operator
2. newInstance() is a method presenting Class "Class", which can be used to create object.
3. If we don't know the class name at the beginning and it's available dynamically Runtime then we should go for newInstance() method.

eg: 1. public class Test

```
{ public static void main(String[] args) throws Exception
{ // take input of class name for which object has to be
  created at the runtime.
```

String className = args[0];

// load class file explicitly

Class c = Class.forName(className);

// for the loaded class object is created using zero param constructor
Object obj = c.newInstance();

// perform type casting to get Student object

```
Student std = (Student) obj;
```

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

{

Args (cmd) : java Test Student.

Class Student

```
{ static {
```

```
System.out.println("Student class file is loading"); }
```

```
public Student()
```

```
{ System.out.println("Student constructor is called"); }
```

{}

Output: Student. class file is loading

Student constructor is called

Student@....

Note:

- * If dynamically provide class name is not available then we will get the Runtime Exception saying "ClassNotFoundException".
- * To use newInstance() method Compulsory Corresponding Class Should contain no Argument Constructor, otherwise we will get the Runtime Exception saying "InstantiationException".
- * If the Argument Constructor is private then it would result in "IllegalAccessException".
- * During typecasting if there is no relationship b/w two classes as parent to child then it would result in "ClassCastException".

Difference between new and newInstance()

new : * new is an operator, which can be used to create an Object.

- * We can use new operator if we know the class name in the beginning
- * If the corresponding class file is not available at the

Runtime then we will get Runtime Exception saying "NoClassDefFoundError". It is unchecked.

To used new Operator the corresponding Class not required to contain no argument constructor.

newInstance() : * newInstance() is a method, present in class "Class" used to create an object

- * We can use the newInstance() method, if we don't write Class name at the beginning and available dynamically Runtime.

- * Object o = Class.forName(args[0]).newInstance();

- * If the Corresponding .class file not available at runtime then we will get runtime exception saying "ClassNotFoundException", it is checked.

- * To used newInstance() method the corresponding class should Compulsory Contain no Runtime Exception saying InstantiationException.

Difference between ClassNotFoundException and NoClassDefFound Error :

1. For hard coded class name at Runtime in the corresponding .class files not available we will get NoClassDefFoundError, which is unchecked

Test t = new Test();

In Runtime Test.class file is not available then we will get NoClassDefFoundError

2. For dynamically provided class name at Runtime, if the corresponding .class files is not available then we will get the Runtime Exception saying "ClassNotFoundException".

Ex: Object o = Class.forName("Test").newInstance();
At runtime if .class file not found we get ClassNotFoundException

Note:

New will Create memory on the heap area

Student: → Jvm will search for .class file in current working directory if found load the file into Method Area.

During the loading of .class file

- Static variable will get memory set with default value
- Static block gets executed

In the heap area, for the required object memory for instance variables is given jvm will set the default values to it

- Execute the instance block if available
- Call the constructor to set the meaningful value to the instance variable

Jvm will give the address of the Object to hashing algorithm which generates the hashCode for the Object and that hashCode will be returned as the reference to the programmer.

Import Statements:Class Test

```
{ public static void main (String args[])
    { ArrayList l = new ArrayList();
}
```

Output: Compile time error (Cannot find symbol)

* We can resolve this problem by using fully qualified name "java.util.ArrayList" l = new java.util.ArrayList(); But problem is we need to use this name everytime and increase length of code and reduce readability.

* We can resolve this problem by using Import Statements

```

Example: import java.util.ArrayList;
class Test{
    public static void main(String[] args)
    { ArrayList t = new ArrayList(); }
}

```

- * Hence whenever we are using import statements it is not required to use fully qualified names, we can use short names directly.
- * This approach decreases length of code and improves readability.

Case 1: Types of import statements

- 1) Explicit class import
- 2) Implicit class import

Explicit Class import:

eg: import java.util.ArrayList;

- * This type of import is highly recommended because it improves the readability of the code.
- * Best suitable for developers where readability is important.

Implicit Class import:

eg: import java.util.*;

- * It is not recommended to use because it reduces the readability of the code.
- * Best suited for students where typing is important.

Case 2 Which of statements are meaningful: ?

- a. import java.util;
- b. import java.util.ArrayList.*;
- c. import java.util.*;
- d. import java.util.ArrayList;

Implicit Import

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

```
ArrayList
```

Compilation Time ↑

Explicit Import

```
import java.util.ArrayList;
```

```
ArrayList
```

Compilation Time ↓

Case 3 Class MyArrayList extends java.util.ArrayList

- * The code compiles fine even though we are not using import statements because we used fully qualified name.
- * Whenever we are using fully qualified name it is not required to use import statement.
- * Similarly whenever we are using import statements it is not required to use fully qualified name

Case 4

```
import java.util.*;
import java.sql.*;
Class Test{
    public static void main (String [] args)
    {
        Date d = new Date();
    }
}
```

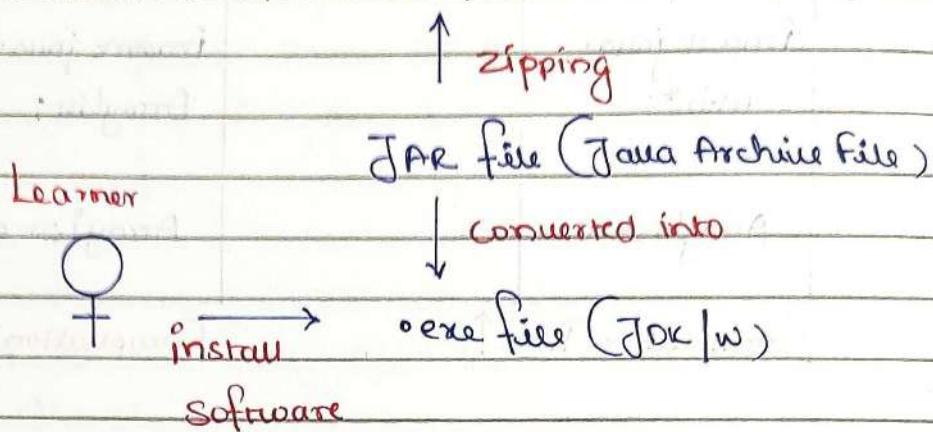
Output: Compile time error

Reference to Date is ambiguous.

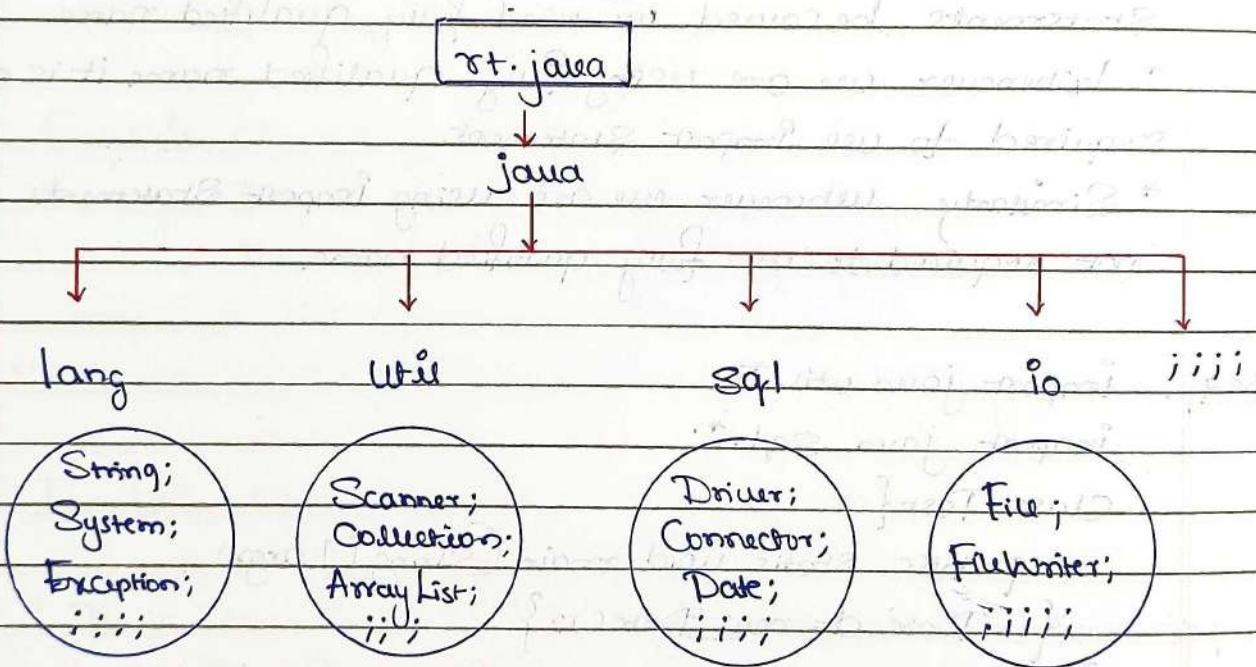
Error in list case also we may get the same ambiguity problem because it is available in both util and sql package.

* JDK software is an API code

API : Collection of .class files (class/interface/enum)



* Sooot .class files are present, for maintenance and easy use they created folder and keep the .class files, these folders are called as "packages" in java.



→ Need to inform compiler and jump about these classes through "import"

Cases

While resolving class name compiler will always give the importance in the following order.

1. Explicit class import

2. Classes present in the current working directory

3. Implicit Class import

eg:

```
import java.util.Date;
import java.sql.*;

class Test{
    public static void main (String args[])
    {
        Date d= new Date();
    }
}
```

The code compiles fine and in this case util package Date will be considered.

Case 6

Whenever we are importing an / a package all classes and interfaces present in that package are by default available but not Sub Package Classes.

java

 |→ util

 |→ Scanner class, ArrayList class, LinkedList class

 |→ regex

 |→ Pattern class

To use Pattern class in our program directly which import statement is required?

Ans

import java.util.regex.*; (implicit import)

import java.regex.Pattern; (explicit import)

Case 7

In any java program the following 2 packages are not required to import because there are available by default to every java program.

1. java.lang package

2. default package (current working directory)

Case 8

"Import Statement is totally Compile-time concept" if more no. of imports are there then more will be compilation but "no change in execution time."

Difference between C language #include and java language import?

	C/C++		Java
C Compiler	#include <stdio.h> 3000+ functions are available	Java Compiler	import java. lang.*; 300+ classes
Static inclusion	Compiler will bring all 3000 functions to our code	Jvm will load System class only when it is needed so we call it as "Dynamic inclusion"	System String StringBuffer
(memory is wasted.)			(Memory is effectively utilised)

#include

1. It can be used in C and C++.
2. At compile time only Compiler copy the code from std library and placed in curr. program.
3. It is static inclusion.
4. wastage of memory.

import

1. It can be used in Java.
2. At runtime Jvm will execute the corresponding std library and use its result in curr. program.
3. It is dynamic inclusion.
4. No wastage of memory.

Note: In the case of C language #include : all the header file will be loaded at the time of include statement hence it follows static loading.

- * But in java import statement number of ".class" will be loaded at the time of import statements in the next lines of code whenever we are using a particular class then only corresponding ".class" file will be loaded.
- Hence it follows "dynamic inclusion" or "load on demand" or "load on fly".

JDK 1.5 Version's new features:

- | | |
|--------------------------------|----------------------------|
| 1. For - Each | 6. Co-varient return types |
| 2. Var-Arg | 7. Annotations |
| 3. Queue | 8. Enums |
| 4. Generics | 9. Static import |
| 5. Autoboxing and AutoUnboxing | 10. String Builder |

Static import:

If was introduced in JDK 1.5 versions . According to Sun microsystems : Static import improves readability of code but according to world wide programming experts Static import creates confusion and reduce readability of the code . Hence if there is no specific requirement it is not recommended to use static import .

Usually we can access static members by using class name but whenever we are using static import it is not required to use class name , we can access methods directly .

Without Static Import : Class Test {

```
public static void main (String [ ] args)
{
    System.out (math.sqrt(4)); // 2
    System.out (math.max (10,20)); } // 20
}
```

With Static import:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;

class Test{
    public static void main(String[] args)
    {
        System.out.println(sqrt(4)); // 2.0
        System.out.println(max(10, 20)); // 20
    }
}
```

eg: class Test{

```
    static String name = "Sachin";
    Test.name.length(); // 6
```

eg

System.out.println()

It is a method of PrintStream Class.
It is a reference of PrintStream Class
It is a class

eg:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
```

Class Test

```
{ public static void main(String[] args)
    {
        System.out.println(Integer.MAX_VALUE);
    }
}
```

Output: Compile-time error

Note: Two packages contain a class or interface with the same name is very rare hence Ambiguity problem is very rare in normal import.

* But 2 classes or interface can contain a method or variable with the same name is very common hence ambiguity.

problem is also very common in static import.

- * While resolving static members compiler will give the precedence in the following order:

1. Current class static members
2. Explicit static import
3. Implicit static import.

eg: `import static java.lang.Integer.MAX_VALUE;`

`import static java.lang.*;`

`Class Test {`

`static int MAX-VALUE = 999;`

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

`{ System.out.println(MAX-VALUE); }`

`}`

Which of the following statements are valid?

`import java.lang.Math.*; // invalid`

`import static java.lang.Math.*; // valid`

`import java.lang.Math; // valid`

`import static java.lang.Math; // invalid`

`import static java.lang.Math.sqrt.*; // invalid`

`import java.lang.Math.Sqrt; // invalid`

`import static java.lang.Math.Sqrt(); // invalid`

`import static java.lang.Math.Sqrt; // valid`

Usage of static import reduce readability and create confusion

Hence if there is no specific requirement never recommend to use static import.

Difference between general import and static import.

Normal import:

* We can use normal imports to import classes and interface of package.

- * Whenever we are using ^{normal} static import we can access class and interface directly by their short name, it is not required to use fully qualified names.

Static Import:

- * We can use static import to import static members of a particular class
- * Whenever we are using static import it is not required to use class name, we can access static members directly.

Functional Interface:

An interface having only one abstract method is a functional interface.

e.g.:

```
@FunctionalInterface // indication to compiler (annotation)
interface Demo
{
    void disp(); }
```

Note: Lambda expressions and Functional Interface, they work together.

Lambda Expression:

In Java 8 a new operator was introduced



— arrow operator / lambda operator to write lambda expression.

e.g. interface Demo
{ void disp(); }

Public class Launch {

 public static void main(St[ta])

<pre>Demo d = () -> System.out.println("Hello");</pre>	<pre>d.disp(); } } </pre>
	bracket for multi. stmts.
	Output: Hello

eg. interface Demo

```
{ public void disp(); }
```

public class Launch {

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

```
{
```

```
    // Demo d = new Demo(); // not valid
```

```
    Demo d = new Demo() { // implementation of functional
```

// Anonymous

method approach.

```
    void disp()
```

```
    { System.out ("Hello"); }
```

```
}
```

```
d.disp();
```

interface without "implements"

// multiple methods can be

implemented like this from a

normal interface.

```
}
```

// you cannot write more than one method in func. interf.

eg. @FunctionalInterface

interface Add

```
{ void add (int a, int b); }
```

@FunctionalInterface interface Sub

```
{ void sub (int num1); }
```

Public Class LaunchLambda

```
Add add = (a, b) → { // implementing Add interface
```

int res = a + b; // writing data type of

System.out (res); }; parameters is optional

```
add.add (10, 20); // 20
```

Sub sub = num1 → { // implementing Sub interface

int res = num1 - 5; // no need of parenthesis

return res; }; for single parameter

/Sub Sub = num1 → { return num1 - 5; }^{Curly brackets req.}

// Valid lambda expression, no need to specify
the data type of parameters,

/Sub Sub = num1 → num1 - 5; // bracket not req. cos no return,
// no need to write return statement, compiler will
automatically detect the return type and return.

Sub. Sub(10); // 5; }

}

Key points about lambda expressions:

1. To write lambda expression we use lambda Operator (\rightarrow).
2. Lambda operator divided into two parts to write lambda exp:
left side of lambda operator we write parameters required.
right side we write body or implementation.
3. Left side for parameters declaring/writing data type is optional.
4. Right side if implementation/body has one statement then {} is optional.
5. Left side if parameter is single the () is optional.
6. Right side in body if its single line implementation then return statement is also optional.
7. {} is mandatory if there are more than one statement and also if there is return statement explicitly used by the developer.

WAP to compute length of String:

@ Functional Interface

interface CLS

{ int getLength (String); }

```
public class Launch {  
    public static void main (String [] str)  
    {  
        CLS s1 = str → str.length(); // lambda expression  
        System.out.println (s1.getLength ("Nuke"));  
    }  
}
```

Output: 5

Exception Handling

Exception handling in java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception handling is a mechanism to handle runtime errors.

Exception is an unwanted or unexpected event which occurs during execution of a program at runtime that disrupts the normal flow of program instructions. Exceptions can be caught and handled by the program. (Default Exception Handler).

Major reasons why an exception occurs:

- * invalid user input
- * device failure
- * loss of network connection
- * Physical limitations
- * Code errors
- * Opening an unavailable file.

When an exception occurs within a method, it creates an object. This object is called exception object. It contains information about the exception, such as the name and description of the exception and state of the program when the exception occurred.

If the exception is not handled by the developer, then the object will go to the "Default Exception Handler" else if try and catch blocks are given by the developer then Jvm will not go to "Default Exception Handler".

Example 1

```
import java.lang.util.*;
```

```
Sysout ("Connection Established");
try {
```

```
Scanner Scan = new Scanner (System.in);
```

```
int num1 = Scan.nextInt();
```

```
int num2 = Scan.nextInt();
```

```
int res = num1 / num2;
```

```
Sysout ("The result is " + res); }
```

Input: 10 5

Output: The result
is 2

```
Catch (Exception e) // General exception
```

```
{ Sysout ("You are dividing by zero"); }
```

Input: 10 0

Output: You are
dividing by zero

Examp. 2

```
Sysout ("Connection established");
```

```
try {
```

```
Scanner Sc = new Scanner (System.in);
```

```
Sysout ("Enter 1st number");
```

```
int num1 = sc.nextInt(); // num1 input
```

```
Sysout ("Enter 2nd number");
```

```
int num2 = sc.nextInt(); // num2 input
```

```
int res = num1 / num2;
```

```
Sysout ("Result is : " + res); // Executes if exception not  
occur
```

```
Sysout ("Enter array size");
```

```
int size = Scan.nextInt(); // size input for array
```

```
int [] a = new int [size]; // array initialization
```

```
Sysout ("Enter element to be inserted");
```

```
int ele = sc.nextInt();
```

```
Sysout ("Enter position"); // position should be from
```

```
int pos = sc.nextInt(); 0-(size-1),
```

$a[pos] = ele;$

Sysout ("Element is inserted"); }

Catch (ArithmeticException ae)

{ Sysout ("Dividing by zero"); }

// Specific catch blocks are executed, rest are not executed

catch (NegativeArraySizeException nae)

{ Sysout ("Please be positive"); }

// "Exception" is the parent of all exceptions is kept

Catch (ArrayIndexOutOfBoundsException a) at last.

{ Sysout ("Be in your limits"); }

Sysout ("connection terminated");

Input:

Connection Established

Enter 1st number 100

Enter 2nd number 0

Dividing by zero

Connection terminated

Test Case 2:

Connection Established

Enter 1st number 50

Enter 2nd number 5

The result is 10

Enter array size 5

Enter element to insert 500

Enter position 5

Be in your limits

Connection terminated

Test Case 3:

Connection established

Enter 1st number 70

Enter 2nd number 7

The result is 10

Enter array size -5

Please be positive

Connection terminated.

Note: * Single try block and multiple catch blocks are allowed.

* Corresponding catch block will be called and rest will not be executed.

* As a developer you need to give generic exception "Exception" for default.

Key points:

- * In java to handle exception we use try and catch blocks
- * Catch block will be executed only if there is an exception in try block otherwise it will not.
- * If there is an exception, then the statements below the statement where the exception has occurred will not be executed.
- * One try block can have multiple catch blocks
- * When we are writing catch blocks specific to every kind of exception, it is recommended to use generic catch block
- * The generic catch block must be at the end

Whenever there is an Exception:

1. Handle exception (try and catch block)
2. Duck the exception (throw)
3. Re-throwing an exception (throw, throws, try, catch, finally)

Exceptions can be Categorized in two ways:

1. Built-in Exceptions: Built in exceptions are the exceptions that are available in Java libraries.
 - (a) Checked exceptions: Checked exceptions are called Compile time exceptions because the exceptions are checked at compile time by the compiler.
 - (b) Unchecked exceptions: The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time.
2. User defined exceptions: Sometimes, the built in exceptions in Java are not able to describe a situation. In such cases users can also create exceptions, which are called 'User-defined exceptions'.

Example

```
import java.util.Scanner;
```

Class Alpha

```
{
    void alpha() throws ArithmeticException
    {
        System.out.println("Connection established"); // throws Keyword throws
        Scanner Scan = Scan.nextInt(); // the exception to whom
        System.out.println("Enter first number");
        int num1 = Scan.nextInt();
        System.out.println("Enter second number"); // caught it and handled
        int num2 = Scan.nextInt();
        int res = num1 / num2;
        System.out.println("The res is " + res);
        System.out.println("Connection is terminated");
    }
}
```

// if catch block is within the method, then exception is handled and main catch is not executed.

Class Beta

```
{
    void beta() throws ArithmeticException
    {
        Alpha a = new Alpha();
        a.alpha();
    }
}
```

public class Launch

```
{
    public static void main(String[] args)
    {
        try
        {
            Beta b = new Beta();
            b.beta();
        }
    }
}
```

Catch (ArithmeticException e)

```
{
    System.out.println(e.getMessage());
    System.out.println("Exception finally handled
        in main");
}
```

Input: 100 0Output:

"Exception finally
handled in main."

Example

```

import java.util.Scanner;
// Re-throwing an exception
class Alpha1
{
    void alpha() throws ArithmeticException
    {
        System.out.print("Connection Established");
        try
        {
            Scanner sc = new Scanner(System.in);
            System.out("Enter first number");
            int num1 = sc.nextInt();
            System.out("Enter second number");
            int num2 = sc.nextInt();
            int res = num1 / num2;
            System.out("The result is " + res);
        }
    }
}

```

Catch (Arithmetic Exception e)

```

{
    System.out.println("Exception handled in Alpha");
    throw e; // rethrowing the exception to the method where
              // Alpha1 was created
}

```

finally

```

{
    System.out.println("Connection is terminated");
}
// finally will execute after the exception is handled
}

```

public class Launch

```

{
    public static void main(String[] args)
    {
        try
        {
            System.out("Main Method");
            Alpha1 a = new Alpha1();
            a.alpha();
        }
    }
}

```

Catch (Arithmetic Exception e)

```

{
    System.out("Exception handled in main");
}

```

Output : Main method

Connection established

Enter first number 100

Enter Second number 0

Exception handled in Alpha

Connection is terminated.

Exception handled in main

throw

throws

- | | |
|---|---|
| * Written inside method, usually in Catch block | Written in method signature |
| * It is used to rethrow an exception | It is used to catch an exception. |
| * Statements below the throw keyword are not executed | Statements below the 'throws' keyword are executed in the method. |

Catch : → if there is no exception, block will not execute

→ Only executes if there is a matching exception

finally : Finally block statements are executed if there are exceptions, no exception, throw keyword, hidden stack.

Purpose of:

try and catch → to handle exception

throws → catch and method signature

throw → Catch → rethrow

finally → Close resource

The Exception Object Contains : 1. Name of the exception

2. Description of the exception

3. Stack trace of the exception

Methods to print exception information:

1. getMessage() : prints the description of the exception

Example: / by zero

2. toString() : prints the name and description of exception

Example: ArithmeticException: / by zero

3. PrintStackTrace() : prints the name and the description of the exception along with stack trace.

Example: ArithmeticException: / by zero at Demo.alpha();

Example

Class Demo

```

{ int disp()
  {
    try
    {
      System.out.println("Method Started");
      return 10; } //finally block cannot be written without
    finally
    {
      System.out.println("Method Ending"); }
  } //finally block will be executed even after return
  statement (finally will dominate "return").
```

public class Launch{

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

```
  {
    Demo d = new Demo();
  }
```

```
  d.disp(); }
```

```
}
```

Output: Method Started

Method Ending.

Note: `System.exit(0)` will dominate finally and return statement.

In above example replacing 'return' with `System.exit(0)` then the program will stop after the first statement and the program is terminated, no more running.

Code Snippets

1. Class Atom

{ Atom() { Sysout("atom"); } }

Class Rock extends Atom

{ Rock (String type) { Sysout(type); } }

public class Mountain extends Rock {

Mountain()

Super ("granite");

new Rock ("granite"); }

public static void main (String [1a]) { new Mountain(); }

}

Output: atom granite atom granite

2. Interface TestA { String toString(); }

public class Test {

public static void main (String [] args)

{ Sysout (new TestA) { public String toString() { return "test"; } } }

{ }

3. Class Building { }

Public class Barn extends Building { }

public static void main (String [] args)

{ Building b1 = new Building(); Barn b1 = new Barn(); }

Barn b2 = (Barn) b1; Object ob1 = (Object) b1;

String str1 = (String) b1; Building b2 = (Building) b1; }

{ ↳ RE: Class Cast Exception (this should be removed to compile).

4. interface Foo { }

Class Alpha implements Foo { }

Class Beta extends Alpha { }

Class Delta extends Beta { }

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

Beta x = new Beta();

16. // insert code here! }

? What line should be inserted to get ClassCastException?

Ans: Foo f = (Delta)x;

5. Public class Batman { int square = 81;

Public static void main (String [] args) { new Batman().go(); }

void go () {

incr (++square); System.out (square); }

void incr (int square) { square += 10; }

? Output: 82

6. Class Pass { public static void main (String [] args) { }

{ int x = 5; Pass p = new Pass();

p.doStuff(); System.out ("main x = " + x); }

void doStuff (int x) { }

{ System.out ("do stuff x = " + x); }

Output: do stuff x = 5

main x = 5

7. String [] elements = { "for", "tea", "too" };

String first = (elements.length > 0) ? elements[0] : null;

8. Class Foo { public int a = 3; // instance variable

public void addFive () { a += 5; System.out ("f"); }

? Class Bar extends Foo { public int a = 8; }

public void addFive () { this.a += 5; System.out ("b"); }

?

Exception Handling in Java

try {

Statement 1;

Statement 2;

Statement 3;

try {

Statement 4;

Statement 5;

Statement 6; }

Catch (xxx e)

{ Statement 7; }

finally { Statement 8; }

Statement 9;

}

Catch (yyy e)

{ Statement 10; }

finally { Statement 11; }

}

Statement 12;

Case 6: Exception at 5, and both
catch blocks not matched.

Statements 8, 4, 1, 2, 3, 11 will be
executed (abnormal termination)

Case 1: If no exception occurs

Statements 1, 2, 3, 4, 5, 6, 8, 9, 11, 12
will be executed.

Case 2: If exception occurs

at Stmt 2 and corresponding catch
block is matched.

Statements 1, 10, 11, 12 will
be executed

Case 3: If exception occurs

at Stmt. 2 and corresponding
catch block is not matched.

Statements 1, 11 will be
executed resulting in abnormal
termination.

Case 4: If exception occurs

at 5, and corresponding
inner block is matched.

Statements 1, 2, 3, 4, 7, 8, 9, 11, 12
will be executed (normal)

Case 5: Exception at 5, and
inner block not matched but
outer catch block matched

Statements 1, 2, 3, 4, 8, 10, 11, 12
will be executed (normal)

Valid Syntax in Exception Handling

1. Only try

`try { ... }` Not Allowed

2. Only Catch

`Catch (xxx e)` Not allowed
`{ ... }`

3. Only finally

`finally { ... }` Not allowed

4. try-Catch

`try { ... }`
`Catch (xxx e)` Allowed
`{ ... }`

5. Reverse order

`Catch (xxx e)`
`{ ... }` Not allowed
`try { }`

6. Multiple try

`try { ... }`
`try { ... }` Not valid
`Catch (xxx e)`
`{ ... }`

7. Multiple try and catch

<code>try { ... }</code>	<code>try { ... }</code>
<code>Catch (xxx e)</code>	<code>Catch (xxx e)</code>
<code>{ ... }</code>	<code>{ ... }</code>

Valid

8. Multiple try

`try { ... }``Catch (xxx e) Not allowed``{ ... }``try { ... }`

9. Multiple Catch

`try { ... }``Catch (xxx e) { ... } Allowed``Catch (yyy e) { ... }`

10. Multiple Catch

`try { ... }``Catch (xxx e) { ... } Not allowed``Catch (xxx e) { ... }`

11. Multiple Catch

`try { ... }``Catch (xxx | yyy e) { ... } Allowed`

12. try-finally

`try { ... } Allowed``finally { ... }`

13. Catch-finally

`Catch (xxx e)``{ ... } Not allowed.``finally { ... }`

14. Unordered

`try { ... }``finally { ... }``Not allowed``Catch (xxx e) { ... }`

15. try - multiple catch - finally

```
try{...}
catch (xxx e){...} Allowed
catch (yyy e){...}
finally {...}
```

16. Multiple finally

```
try{...}
catch (xxx e){...} Not allowed
finally {...}
finally {...}
```

17. Statements in between try - Catch - finally

```
try{...}
System.out ("");
catch (yyy e){...}
System.out ("");
finally {...}
```

18. Only try with resource

```
try (R)
{ ...}
```

Synchronous Exceptions :- Occur at specific program statement.

Class Launch

```
{ public static void main (String [] args)
{
    String Str= null;
    Str.toUpperCase();
```

```
}
```

→ Null Pointer Exception

A Synchronous Exception: occurs anywhere in the program.

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

```
    Scanner Scan = new Scanner (System.in);
```

```
    System.out ("Enter name");
```

```
    String name = Scan.next();
```

```
    System.out ("Enter your grades");
```

```
    String grade = Scan.next(); }
```

Output: Enter your name: Sachin

Enter your grades: **CTRL + C** (Keyboard interrupt)

Exception in thread "main" Terminate Batch Job (Y/N)?

User Defined Exception

Code

```
import java.util.Scanner;
```

```
class InvalidCustomerException extends Exception // User defined
{ public InvalidCustomerException (String msg) exception
    { super (msg); }}
```

Class Atm

```
{ int userId = 1212; // data for verification
```

```
int password = 1111;
```

```
int pw; int uid;
```

```
public void input()
```

```
{ Scanner Scan = new Scanner (System.in);
```

```
System.out ("Kindly enter user id");
```

```
uid = Scan.nextInt();
```

```
Sysout ("Enter the password");
pw = Scan.nextInt(); }
```

```
{ Public void verify () throws InvalidCustomerException
{ if (Cuserid == uid) && (password == pw)
{ Sysout ("Take your cash"); }
else
{ InvalidCustomerException ice = new InvalidCustomerException ("Are you sure?"); // passing args to
Sysout (ice.getMessage()); // Constructor
throw ice; } // re-throwing
}
}
```

Class Bank

```
{ public void initiate ()
{ Atm a = new Atm();
try {
a.input();
a.verify();
}
Catch (InvalidCustomerException e1) // first catch block
{
try { a.input(); // re-enter the data for verification
a.verify(); }
}
}
```

Catch (InvalidCustomerException e2)

```
{ try {
a.input(); // second catch block
a.verify(); // re-confirmation
}
}
// final time verification
```

```
Catch (InvalidCustomerException e3) // final catch block
{
    System.out.println("We caught you! Card is blocked");
    System.exit(0); // terminate the program.
}
```

Public Class LaunchCE

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

1

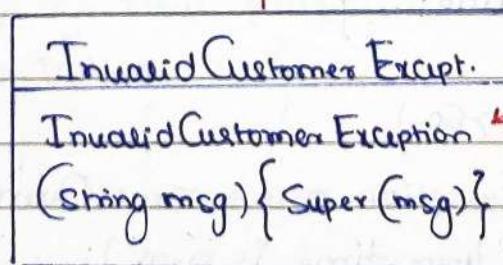
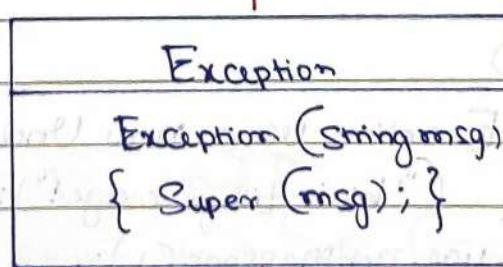
```
Bank b = new Bank();
```

```
bi.initiate(); }
```

`detailMessage` is used by these methods to print exception info.

```
Throwable
String detailMessage;
Throwable (String msg)
{ detailMessage = msg; }

getMessage()
toString()
PrintStackTrace()
```



A simple RIO program to implement exception handling:

```
import java.util.Scanner;  
class UnderAgeException extends Exception // user defined  
{  
    public UnderAgeException (String msg) Class 1  
    {  
        super (msg); }  
}
```

```
Class OverageException extends Exception // user defined  
{  
    public OverageException (String msg) Class 2  
    {  
        super (msg); }  
}
```

Class Applicant

```
{  
    int age;  
    public void input()  
    {  
        Scanner Scan = new Scanner (System.in);  
        System.out ("Enter your age");  
        age = Scan.nextInt(); }  
}
```

```
void verify() throws UnderAgeException, OverageException  
{  
    if (age < 18)  
    {  
        UnderAgeException ue = new UnderAgeException  
        ("Wait till your age!");  
        System.out (ue.getMessage());  
        throw ue; }  
}
```

else if (age > 60)

```
{  
    OverageException oe = new OverageException  
    ("your time is near"); }
```

```
Sysout (oae.getmessage(1));
-throw oae; }
```

```
else { Sysout (" You are eligible"); }
}
```

Class RTO

```
{ public void initiate()
{ Applicant a = new Applicant();
try{ a.input();
a.verify(); } }
```

Catch (UnderAge Exception | Overage Exception e) // handle one
{ try{ a.input();
a.verify(); }

Catch (UnderAge Exception | Overage Exception e1)

```
{ Sysout ("Don't ever try again");
System.exit (0); } // terminate the program.
```

Public Class LaunchRTO

```
public static void main (String [] args)
{ RTO r= new RTO ();
r.initiate(); }
```

JDK 1.7 version enhancements

1. -try with resource
2. -try with multicatch block

Until jdk 1.6, it is compulsory required to write finally block to close all the resources which are open as a part of try block.

```
eg:: BufferedReader br = null;  
try{  
    br = new BufferedReader(new FileReader("abc.txt"));  
}  
catch (IOException ie)  
{  
    ie.printStackTrace();  
}  
finally{  
    try{ if(br != null)  
        br.close(); }  
    catch (IOException ie)  
    { ie.printStackTrace(); }  
}
```

Problems in this approach:

1. Compulsorily the programmer is required to close all opened resources which increases the complexity of the program.
2. Compulsorily we should write finally block explicitly, which increases the length of the code and reduces readability.

To overcome this problem SUN MicroSystem introduced try with resources in "1.7" version of Jdk.

try with resource

In this approach, the resources which are opened as a part of try block will be closed automatically once the control reaches to the end of the code try block normally or abnormally, so it is not required to close explicitly and complexity of the program would be reduced.

It is not required to write finally block explicitly, so length of code would be reduced and readability is improved.

eg:: try (BufferedReader br = new BufferedReader (new FileReader ("abc.txt")))

{ // use br and perform the necessary operation
 // Once Control reaches to the end of try block automatically
 // br will be closed }

Catch (IOException ie)

{ // handling code }

Rules of using try with resource

1. We can declare any no. of resource, but all these resources should be separated with ;

eg:: try (R₁; R₂; R₃;) {
 // Use the resource }

2. All resources are said to be AutoClosable resources if the class implements an interface called as "java.lang.AutoCloseable". either directly or indirectly.

eg:: java.io package classes, java.sql package classes.
 public interface java.lang.AutoCloseable {
 public abstract void close() throws java.lang.Exception; }

Note: Which ever class has implemented this interface those class objects are referred as "resources".

3. All resource reference by default are treated as implicitly final and hence we can't perform assignment within try block.

```
try (BufferedReader br = new BufferedReader (new FileWriter
("abc.txt")))
{
    br = new BufferedReader (new FileWriter ("abc.txt"));
}
```

Output :: CE Can't reassign a value.

4. Until 1.6 version try should compulsorily be followed by either catch or finally, but from 1.7 version we can take only "try with resource" without catch or finally.

```
try (R) {
    /valid }
```

5. Advantage of try with resource concept is finally block will become dummy because it is not required to close resource explicitly.

6. try with resource nesting is also possible

```
try (R1) {
    try (R2) {
        try (R3) {
            / }
        }
    }
}
```

Note: A code which should repeat multiple times in our project with small change or no change, such code we call it as "boilerplate" code.

Multi Catch Block

Till jdk 1.6, even though we have multiple exception having same handling code we have to write a separate catch block for every exception, it increases length of code and reduces readability.

eg:: try{...
...}

Catch (ArithmeticException ae){
ae.printStackTrace(); }

Catch (NullPointerException ne){
ne.printStackTrace(); }

To overcome this problem Sun has introduced "multi catch block" concept in 1.7 version

eg:: try{...
...}

Catch (ArithmeticException | NullPointerException ae)
{ e.printStackTrace(); }

Catch (ClassCastException | IOException e)
{ e.printStackTrace(); }

In multi catch block, there should not be any relation b/w exception types (either Child to parent or parent to child or same type) it would result in compile time error.

eg:: try{..}

Catch (ArithmeticException | IOException)
{ e.printStackTrace(); }

Output: CE

Rules of overriding when exception is involved.

While overriding if the child class method throws any checked exception compulsorily the parent class method should throw the same checked exception or its parent. Otherwise we will get compile time error.

There are no restrictions on unchecked exceptions.
eg::

```
Class Parent { public void m1(); }

Class Child extends Parent
{ public void m1() throws Exception {} }
```

Error: m1() in Child cannot override m1() in Parent
public void m1() throws Exception {}
Overridden method does not throw Exception.

Rules w.r.t Overriding

1. Parent: public void m1() throws Exception {}

Child: public void m1()

Valid

2. Parent: public void m1() {}

Child: public void m1() throws Exception {}

Invalid

3. Parent: public void m1() throws Exception {}

Child: public void m1() throws Exception {}

Valid

4. Parent: public void m1() throws IOException {}

Child: public void m1() throws IOException {} **Valid**

5. Parent: public void mi() throws IOException {}
 Child: public void mi() throws FileNotFoundException,
 EOFException {} **Valid**
6. Parent: public void mi() throws IOException {}
 Child: public void mi() throws FileNotFoundException,
 InterruptException {} **Invalid**
7. Parent: public void mi()
 Child : public void mi() throws ArithmeticException,
 NullPointerException, RuntimeException {} **Valid**
8. Parent: public void mi() throws IOException
 Child : public void mi() throws Exception {} **Invalid**
9. Parent: public void mi() throws Throwable {}
 Child : public void mi() throws IOException {} **Valid**

* instanceof : We can use the instanceof operator to check whether the given an object is particular type or not

eg:: $\text{reference instanceof Class/Interface\ name}$

eg:: ArrayList al = new ArrayList(); // inbuilt object where we can keep any type of other objects
 al.add(new Student()); // 0th position
 al.add(new Cricketer()); // 1st position
 al.add(new Customer()); // 2nd position

Object o = l.get(0); // l is an arraylist object.

if (o instanceof Student) { Student s = (Student)o;
 // perform Student Specific opn. }

```

        elseif (o instanceof Customer) {
            Customer c = (Customer)o;
            // perform customer specific operations
        }
    
```

eg:: Thread t = new Thread();
 Sysout (t instanceof Thread); // true
 Sysout (t instanceof Object); // true
 Sysout (t instanceof Runnable); // true

eg:: Public class Thread extends Object implements Runnable { }

* To use instanceof operator compulsorily there should be some relation between argument type (either child to parent or parent to child or same type) otherwise we will get Compile time error saying incompatible types.

eg:: String s = new String ("Sachin");
 Sysout (s instanceof Thread); // CE

* Whenever we are checking the parent object is child type or not by using instanceof operator that we get false.

```

Object o = new Object();
Sysout (o instanceof String); // false
Object o = new String ("abck");
Sysout (o instanceof String); // true
    
```

* for any class or interface x null instanceof x is always returns false Sysout (null instanceof x); // false

eg:: Object t = new Thread();

Sysout (t instanceof Object); // true

Sysout (t instanceof Thread); // true

Sysout (t instanceof Runnable); // true

Sysout (t instanceof String); // false

Sysout (new instanceof Object); // false

- * isInstance(): The isInstance() method is used to check if the specified object is compatible to be assigned to the instance of this class.

Difference between instanceof and isInstance():

instanceof:

- * instanceof is an operator which can be used to check whether the given object is a particular type or not.
- * eg:: String s = new String("Sachin");
Sysout (s instanceof Object); // true

isInstance():

- * isInstance() is a method, present in class "Class", we can use it to check whether the given object is particular type or not. We don't know the type at the beginning, it is available dynamically at runtime.

* eg:: class Test{

```
public static void main (String [ ] args)
{ Test t = new Test();
```

Sysout (Class.forName (args[0]).isInstance (t)); }

}

java Test Test // true

java Test String // false

Code Snippets

1.

```
interface Foo{}  
Class Alpha implements Foo{}  
Class Beta extends Alpha{}  
Class Delta extends Beta{}  
public static void main(String [] args)  
{  
    Beta x = new Beta();  
    // insert code here  
}
```

What code will cause java.lang.ClassCastException?

Foo f = (Delta)x; (Because Foo is not related to Delta)

2.

```
public class Batman{  
    int square=81;  
    public static void main (String [] args)  
    {  
        new Batman().go();  
    }  
    void go(){  
        incr (++square);  
        System.out.println(square);  
    }  
    void incr (int square){ square+=10; }  
}  
Output: 82
```

3.

```
public class Pass{  
    public static void main (String [] args)  
    {  
        int x=5;  
        Pass p = new Pass();  
        p.doStuff(x); System.out.println("main x" + x);  
        void doStuff (int x) // main x=5  
        {  
            System.out.println("doStuff x=" + x++);  
        } // doStuff x=5.  
    }  
}
```

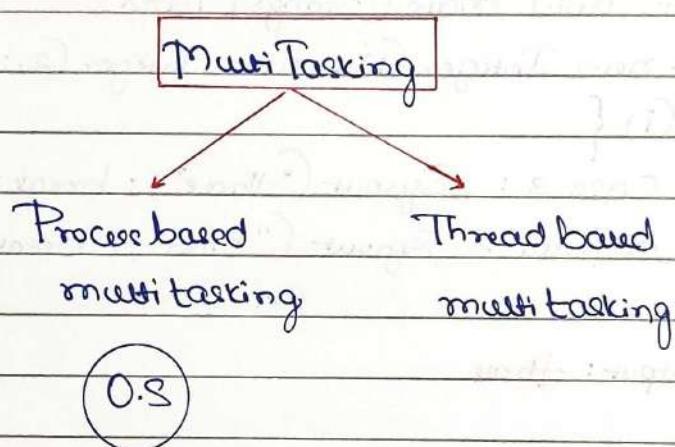
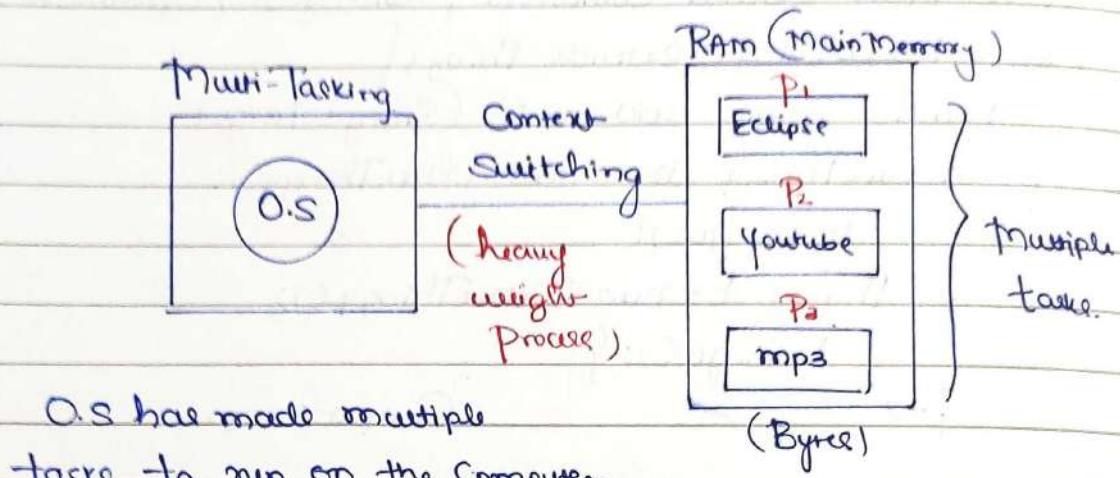
4. Class Thingy { Meter m = new Meter(); }
 Class Component { void go() { System.out("C"); } }
 Class Meter extends Component { void go() { System.out("m"); } }
 Class DeluxeThingy extends Thingy {
 public static void main (String [] args)
 { DeluxeThingy dt = new DeluxeThingy();
 dt.m.go();
 Thingy t = new DeluxeThingy();
 t.m.go();
 } } Output: mm

5. public static void main (String [] args)
 { Integer i = new Integer (1) + new Integer (2);
 switch (i) {
 case 3: System.out("three"); break;
 default: System.out("other"); break; } }
 Output: three

6. Class Mammal {}
 Class Raccoon extends Mammal {} // Raccoon IS-A mammal
 Mammal m = new Mammal(); } Raccoon HAS-A mammal
 Class BabyRaccoon extends Mammal {}
 / BabyRaccoon IS-A mammal, BabyRaccoon HAS-A mammal
 does not have.

7. Public class Hello
 { String title;
 int value;
 public Hello()
 { title += "World"; } }
 public Hello (int value)
 { this.value = value;
 this.title = "Hello";
 Hello(); } }
 Hello c = new Hello();
 System.out(c.title);
 Constructor can't be called explicitly
 Compile Time Error

Multi-threading in Java.



Multitasking: Executing several tasks simultaneously is the concept of multitasking.

There are two types of multitasking:

- Process based multitasking
- Thread based multitasking

Process Based Multitasking: Executing several tasks simultaneously where each task is a separate independent process. Such type of multitasking is called "process based multitasking".

e.g.: Listening to music, watching youtube
Process based multitasking is best suited at O.S level.

Thread based multitasking: Executing several tasks simultaneously where each task is a separate independent part of the same program, is called "Thread based multitasking". Each independent part is called "Thread".

1. This type of multitasking is best suited at 'Programmatic level'. The main advantages of multitasking is to reduce the response time of the system and to improve the performance.
2. The main important application areas of multithreading are
 - a. To implement multimedia graphics
 - b. To develop web applications servers
 - c. To develop video games
 - d. To develop animations
3. Java provides built-in support to work with threads through API called Thread, Runnable, ThreadGroup, ThreadLocal...
4. To work with multithreading, java developer will code only for 10% remaining 90% java API will take care.

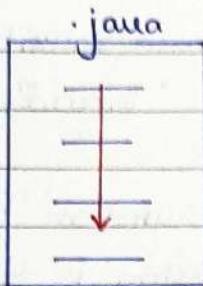
Q5. What is a thread?

Separate flow of execution is called "Thread". If there is only one flow then it is called "Single-thread programming". For every thread there would be a separate job.

In java we can define a thread in 2 ways:

- a. Implementing runnable interface
- b. Extending thread class.

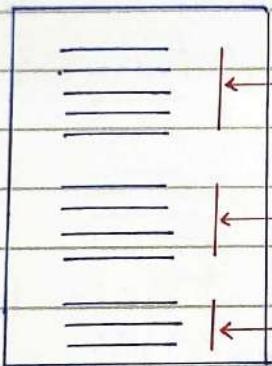
JVM - Single thread



More the waiting time, performance of the application would decrease.

JVM - (threads are light weight)

Multiple-Thread / Multi Threading



T.S

Thread Schedule

It divide the time to all the tasks.

* Less the response time

from application, more would be the performance.

* To utilize CPU time

effectively in our application

* To use multiple task and each task assign to one thread and promote "multithreading".

Software

O.S

Context

Switching

Harddisk

.java

.mp3

.mp4

Ram

3 min

.java

t₁ t₂ t₃

JVM (T.S)

3 min

.mp

4

2 min

.mp3

(Electronics)

Clockcycles (Hz)

Microprocessor

CPU

Time allocation
of every process
dedicated by
O.S

Extending Thread Class

We can create a thread by extending a Thread.

Class MyThread extends Thread {

@Override

```
public void run() {
    for (int i=0; i<2; i++)
        { System.out.println("Child Thread"); }
}
```

{

→ //Defining a thread (writing a class and extending a thread)

→ //job a thread (code written inside run())

Class ThreadDemo {

```
public static void main(String[] args) {
    MyThread t = new MyThread(); // Thread initialization
    t.start(); // Starting a thread
```

//at this line 2 threads are there

for (int i=0; i<2; i++)

```
{ System.out.println("main Thread"); }
```

{}

Behind the scenes:

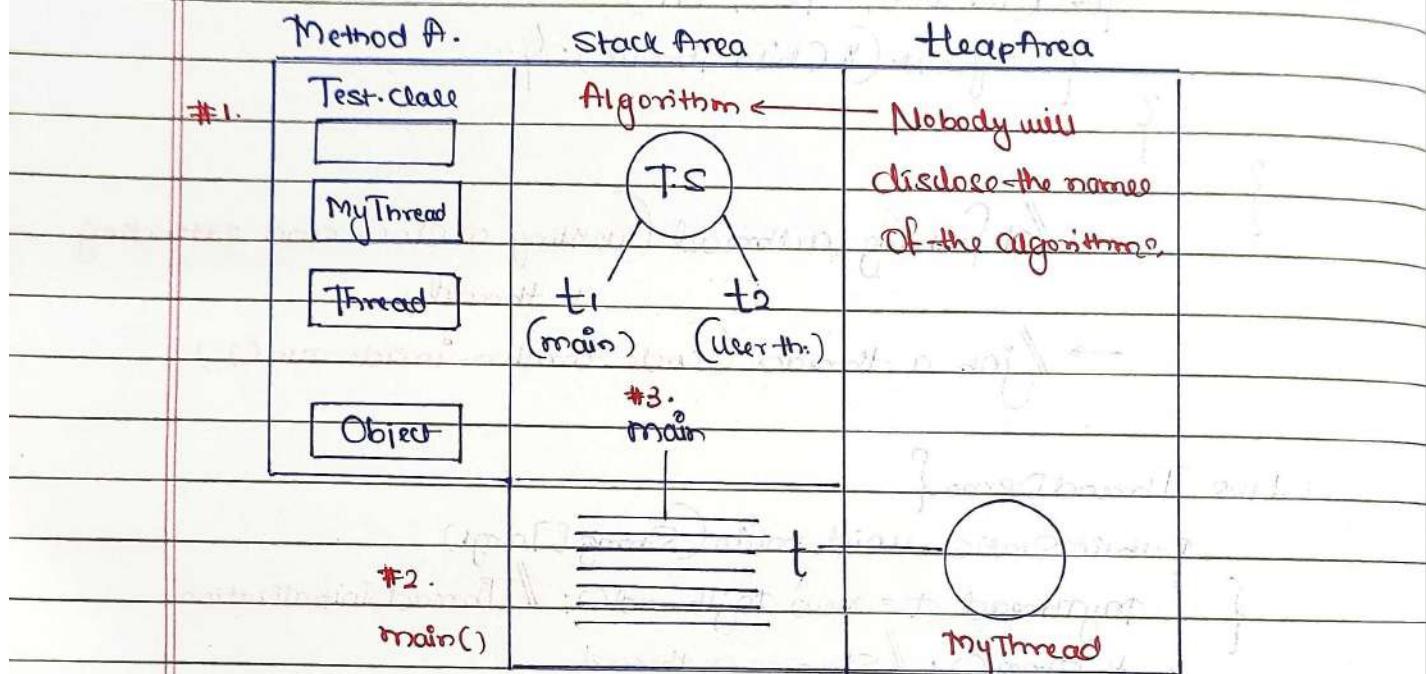
1. Main thread is created automatically by Jvm

2. Main thread creates Child thread and starts the child thread

Thread Scheduler:

If multiple threads are waiting to execute, then which thread will execute first is decided by the Thread Scheduler which is a part of Jvm.

- * In case of multithreading we can't predict the exact output. Only possible output we can expect.
- * Since jobs of threads are important, we are not interested in the order of execution it should just execute such that performance should be improved.



Case 2: Difference between t.start() and t.run()

- * If we call t.start() a separate thread will be created which is responsible to execute run() method.
- * If we call t.run(), no separate thread will be created rather the method will be called just like normal method by main thread.
- * If we replace t.start() with t.run(), then output of program would be:

Child thread

Child thread

main thread

main thread

Case 3:: Importance of Thread class Start() method

For every thread, required mandatory activities like registering the thread with ThreadScheduler will be taken care by Thread Class Start() method and programmer is responsible of just doing the job of the Thread inside run() method.

Start() acts like an assistance to programmer.

```
public void start()
{
    register Thread with ThreadScheduler
    All other mandatory low level activities
    invoke / calling run() method }
```

We can conclude that without executing Thread Class Start() method there is no chance of starting a new Thread in java.

Due to this Start() is considered as heart of multithreading.

Case 4:: If we are not overriding run() method

If we are not overriding run method then Thread class run method will be executed which has empty implementation and hence we won't get any output.

eg::

```
class MyThread extends Thread {}
class ThreadDemo {
    public static void main (String [] args)
    {
        MyThread t = new Thread();
        t.start();
    }
}
```

It is highly recommended to override run() method, otherwise don't go for multi-threading.

Case 5 :: Overloading of run() method

We can overload run method but Thread class start() will always call run() with zero argument. If we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

eg::

```
Class MyThread extends Thread {
    public void run() {
        System.out.println("no arg method");
    }
}
```

```
public void run (int i) {
    System.out.println("zero arg method");
}
```

```
Class ThreadDemo {
```

```
    public static void main (String ... args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Output: no arg method

Case 6 :: Overriding of start() method

If we override start() then our start() method will be executed just like normal method, but no new thread will be created and no new Thread will be started.

eg:: Class MyThread extends Thread {

```
    public void run() {
        System.out.println("no arg method");
    }
}
```

```
    public void start ()
```

```
    {
        System.out.println("start arg method");
    }
}
```

Class ThreadDemo {

```
    public static void main (String ... args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Output: Start arg method.

It is never recommended to override start() method.

Case 7 ::

Class MyThread extends Thread {

```
    public void start()
    {
        super.start();
        System.out("Start method");
    }
}
```

```
    public void run()
    {
        System.out("run method");
    }
}
```

Class ThreadDemo {

```
    public static void main (String ... args)
    {
        MyThread t = new MyThread();
        t.start();
        System.out("main method");
    }
}
```

Output: Main Thread : main method

Start method

User Defined Thread : run method

Case 8 :: Life cycle of a thread

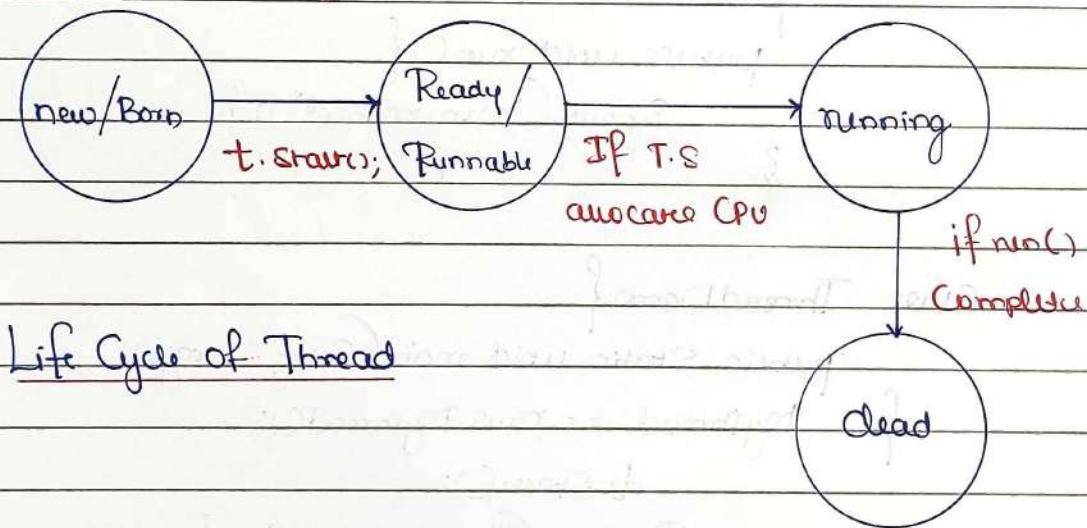
MyThread t = new MyThread(); // Thread is in born state

t.start(); // Thread is in ready/runnable state

If thread Scheduler allocates CPU time then we say thread entered into running state. If run() is completed by thread then we say thread entered into dead state.

- * Once we created a Thread Object then the Thread is said to be in new state or born state.
- * Once we call start() method then the Thread will be entered into Ready or Runnable State.
- * If the Thread Scheduler allocates CPU then the Thread will be entering / entered into running state.
- * Once run() method complete then the Thread will enter into dead state.

MyThread = new MyThread();



Case a::

After starting the thread, we are not supposed to start the same Thread again, then we say Thread is in "Illegal Thread State Exception".

MyThread t = new MyThread(); // Thread is in born state

t.start(); // Thread is in ready state

t.start(); // Illegal Thread State Exception

Creation of Thread Using Runnable interface.

1. Creating a thread using `java.lang.Thread` class.
 - a. Use `start()` from Thread Class.
 - b. Override `run()` and define job of thread.
2. Creation of a Thread requirement to Sunms is an `Sec` interface `Runnable`

```
void run();
```

Class Thread implements Runnable { //Adaptor Class

```
public void start() {
    1. Register the thread with ThreadScheduler
    2. All other mandatory low level activities (memory
       level)
    3. invoke or call run() method. }
```

```
public void run() {
    job for a thread; }
```

Shortcuts of eclipse:

- * `Ctrl + Shift + T` :- To open a definition of any class
- * `Ctrl + O` :- To list all the methods of a class.

Note:

`public java.lang.Thread();`
 → **thread class `start()`, followed by thread class `run()`**

`public java.lang.Thread (java.lang.Runnable);`
 → **thread class `start()`, followed by implementation class of
 Runnable `run()`**

Defining a Thread by implementing Runnable Interface

```
public interface Runnable {
    public void abstract void main(); }
```

```
public class Thread implements Runnable {
```

```
    public void start() {
```

1. register thread with Thread Scheduler

2. All other mandatory low level activities

3. invoke run()

```
    public void run() {
```

//empty implementation

```
}
```

eg::

```
Class MyRunnable implements Runnable {
```

@ Override

```
    public void run() {
```

System.out.println("child thread"); }

```
}
```

```
public class ThreadDemo {
```

```
    public static void main (String... args) {
```

```
        MyRunnable r = new MyRunnable ();
```

```
        Thread t = new Thread (r); // Call MyRunnable run()
```

```
        t.start();
```

```
        System.out.println("main thread"); }
```

```
}
```

Output: main thread

child thread

Case Studies:

MyRunnable r = new MyRunnable();

Thread t1 = new Thread();

Thread t2 = new Thread(r);

Case 1: t1.start();

A new thread will be created, which is responsible for executing Thread class run()

Output: main thread : main thread

Case 2: t2.start();

A new thread will be created, which is responsible for executing MyRunnable run()

Output: main thread : main thread

User defined thread : User thread

Case 3: t1.run();

No new thread will be created, but Thread class run() will be executed just like normal method call.

Output: main thread : main thread

Case 4: t2.run();

No new thread will be created, but MyRunnable class run() will be executed just like normal method call.

Output: main thread : Child thread

main thread.

Case 5: r.start();

It results in Compile Time Error.

Case 6: r.run();

No new thread will be created, MyRunnable class run()

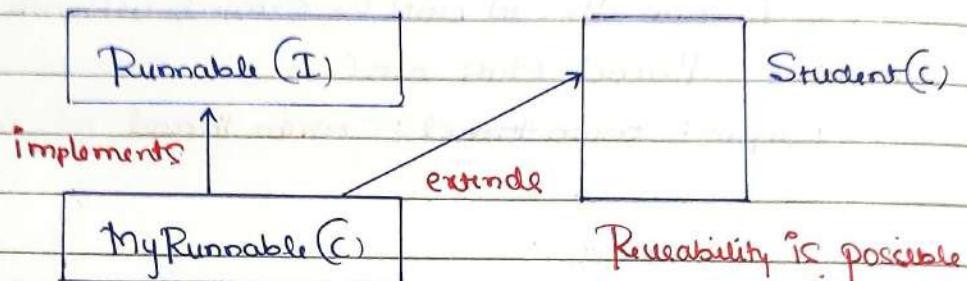
will be executed just like normal method call.

Output: main-thread: child-thread

main-thread

[Good Approach]

1st Approach: implementing Runnable



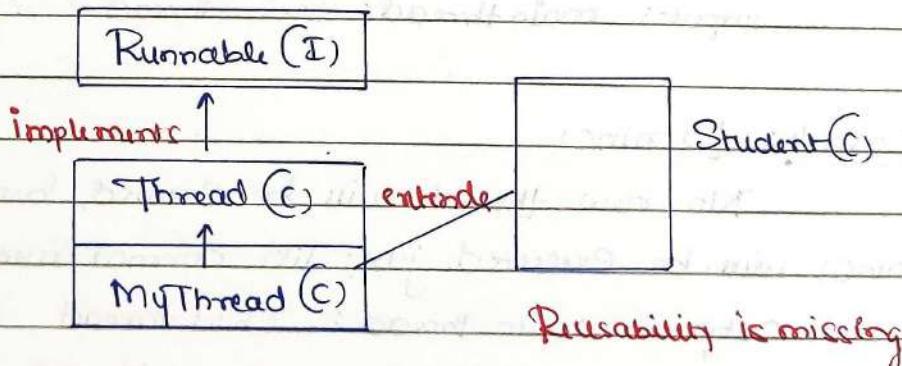
MyRunnable r = new MyRunnable();

Thread t = new Thread(r);

t.start();

[Not Suggested]

2nd Approach: for Extending Thread Class



MyThread t = new MyThread();

t.start();

Q Which approach is the best approach?

- Ane
- * Implementing runnable interface is recommended because our class can extend other class through which inheritance benefit can be brought into our class. Internally performance and memory level is also good when we work with interface.
 - * If we work with extend feature then we will miss all the inheritance benefit because already our class has inherited the feature from "Thread Class", so we normally work prefer extend approach rather implement approach is used in real-time for working with "MultiThreading".

Various Constructors available in Thread Class.

- a. Thread ()
- b. Thread (Runnable r)
- c. Thread (String name)
- d. Thread (Runnable r, String name)
- e. Thread (ThreadGroup g, String name)
- f. Thread (ThreadGroup g, Runnable r)
- g. Thread (ThreadGroup g, Runnable r, String name)
- h. Thread (ThreadGroup g, Runnable r, String name, long stackSize)

Alternate approach to define a thread (not recommended)

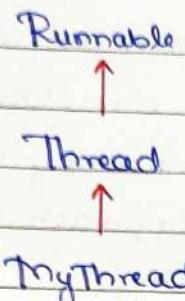
```
Class MyThread extends Thread{
    public void run(){
        System.out.println("Child Thread");
    }
}
```

```
Class ThreadDemo{
    public static void main(String []args){
        MyThread t = new MyThread();
        Thread t1 = new Thread(t);
        t1.start(); System.out.println("Main Thread");
    }
}
```

Output : Main Thread : main thread

Child Thread : Child thread

Internally related



Names of the Thread

Internally for every thread, there would be a name for the thread.

- a. name given by jvm
- b. name given by user.

eg:: class MyThread extends Thread {
 }

```

public class TestApp{
    public static void main(String... args)
    {
        System.out.println(Thread.currentThread().getName()); // main
    }
}
    
```

```

MyThread t = new MyThread();
t.start();
System.out.println(t.getName()); // Thread-0
    
```

```

Thread.currentThread().setName("Yash");
System.out.println(Thread.currentThread().getName()); // Yash
    
```

```

System.out.println(); // Exception in thread "Yash" java.lang...
}
    
```

- * It is possible to change the name of the Thread using `setName()`.
- * It is possible to get the name of the thread using `getName()`.

eg::

Class MyThread extends Thread {

@Override

public void run() {

System.out.println("run() executed by " + Thread.currentThread().getName()); }

public class TestApp {

public static void main(String[] args) {

MyThread t = new MyThread();
t.start();

System.out.println("main() executed by " + Thread.currentThread().getName()); }

Output: run() executed by Thread: Thread-0

main() executed by Thread: main

Code Snippets:

1. public static void test(String str)
{ int check = 4;
if (check == str.length())
System.out.println(str.charAt(check - 1));
else System.out.println(str.charAt(0)); }

And the invocation:
test("four");
test("tree");
test("two");

Output: Compilation fails because no boolean value in "if statement".

2. public interface A { public void m1(); }
Class B implements A {} //CE

Class C implements A { public void m1() {} }

Class D implements A { public void m1(int a) {} } //CE

Abstract Class E implements A { }

Abstract Class F implements A { public void m1() { } }

Abstract Class G implements A { public void m1(int x) { } }

Exactly 2 classes do not compile.

3. Class TestA {

```
public void start() { System.out("TestA"); }
```

```
}
```

public class TestB extends TestA { }

```
public void start() { System.out("TestB"); }
```

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

```
{ ((TestA) new TestB()).start(); }
```

Output: TestB

4. Class Line {

```
public class Point { public int x, y; }
```

```
public Point getPoint()
```

```
{ return new Point(); }
```

```
}
```

Class Triangle {

```
public Triangle()
```

```
// insert code here }
```

```
}
```

In which code when inserted,
retrieve a local instance of
a Point Object?

Line.Point p = (new Line().
getPoint())

5. Public Class Breaker {

```
static String o = "";
```

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

```
{ z: o = o + 2;
```

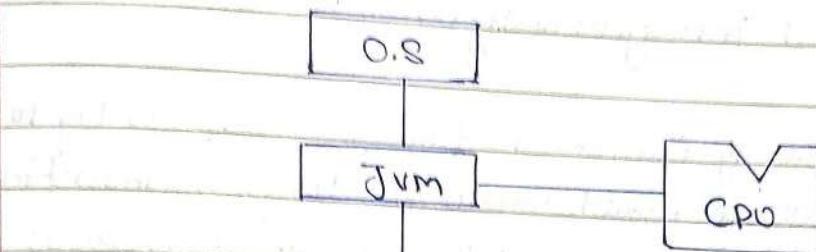
```
for (int x = 3; x < 8; x++) {
```

```
if (x == 4) break;
```

```
o = o + x; } System.out(o); }
```

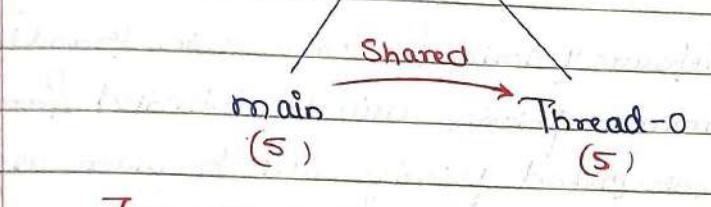
// Compilation fails

Thread Priorities

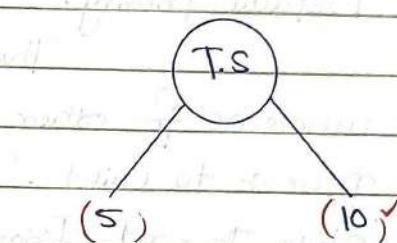


Baudon prio →
-ivity it will decide

if both threads have same priority, then OS
use algorithm which is vendor dependent.



Jvm will create main
thread and it only starts
with a default priority "5".



High priority, Cpu time will
be allocated

For every thread in java has some priority. Valid range of priority is 1-10. If we try to give a different value then it would result in "Illegal Argument Exception".

Thread.MIN-PRIORITY = 1

Thread.MAX-PRIORITY = 10

Thread.NORM-PRIORITY = 5

Thread class does not have priorities is Thread.LOW-PRIORITY, Thread.HIGH-PRIORITY.

Thread Scheduler allocates time (Cputime) based on "priority". If both the threads have the same priority then which thread will get a chance as a Pgm we can't predict because it is vendor dependent.

We can set and get Priority value of the thread using the following methods:

- a. public final void setPriority (int num)
- b. public final int getPriority ()

The allowed priority number is from 1-10, if we try to give other value it would result in "Illegal Argument Exception".
 System.out.print(Thread.currentThread().setPriority(100));
// Illegal Argument Exception.

Default Priority:

The default priority for only main thread is 5; whereas for other threads priority will be inherited from Parent to Child. Parent thread priority will be given as Child Thread Priority.

```
eg:: Class MyThread extends Thread {}  

  public class TestApp{  

    public static void main(String[] args)  

    {  

      System.out.println(Thread.currentThread().getPriority());  

      Thread.currentThread().setPriority(7);  

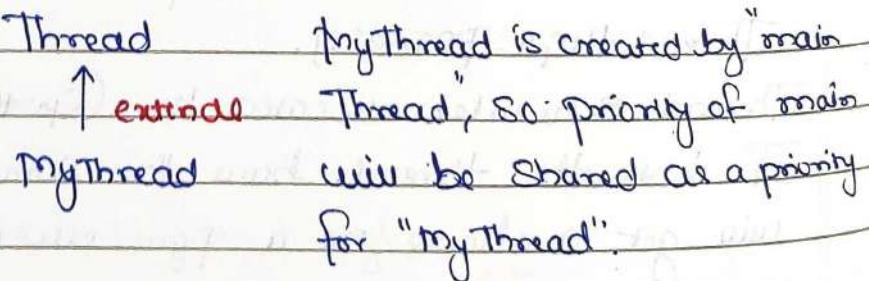
      MyThread t = new MyThread();  

      System.out.println(Thread.currentThread().getPriority());  

    }  

  }
```

Reference:



```
eg:: Class MyThread extends Thread {  

  @Override  

  public void run()
```

classmate
Date _____
Page _____

```

    {
        for (int i=0; i<5; i++) {
            System.out.println("child-thread");
        }
    }

    public class TestApp {
        public static void main(String[] args) {
            MyThread t = new MyThread();
            t.setPriority(7); //line1
            t.start();
            for (int i=0; i<5; i++) {
                System.out.println("main-thread");
            }
        }
    }

```

Since Priority of child-thread is more than main-thread, JVM will execute child thread first whereas for the parent thread Priority is 5, so it will get last chance. If we comment "line-1", then we can't predict the order of execution because both the threads have the same priority.

Some platforms won't provide proper support for Thread priority.
eg.: Windows 7, Windows 10..

We Can prevent Thread from Execution:

- a. yield()
- b. sleep()
- c. join()

yield() : * It causes to pause current executing Thread for giving chance for waiting threads of same priority.
 * If there is no waiting Thread or all waiting threads have low priority then same Thread can continue its execution.
 * If all the threads have same priority and if they are waiting then which thread will get a chance we can't expect; it depends on Thread Schedule.

The Thread which is yielded, when it will get the chance once again depends on the mercy of "Thread Scheduler" and we can't expect exactly.

→ public static native void yield()

MyThread t = new MyThread() // new state

t.start // enter into ready/runnable state

* If Thread Scheduler allocate processor then enters into running state.

- if running class Thread calls yield() then it enters into runnable state.

* If run is finished with execution then it enters into dead state.

eg:: class MyThread extends Thread {

@Override

public void run() {

for (int i=1; i<=5; i++) {

System.out.println("Child thread");

Thread.yield(); } // Line 1

}

public class TestApp {

public static void main (String... args) {

MyThread t = new MyThread();

t.start();

for (int i=1; i<=5; i++)

System.out.println("parent thread"); }

}

Note: If we comment line-1, then we can't expect the output because both the threads have same priority thus which Thread - the ThreadScheduler will schedule is not in the hands of programmer but if we don't comment line-1, then there is a possibility of main thread getting more no. of times, so main thread execution is faster than child thread will get a chance.

Some platforms won't provide proper support for `yield()`, because it is getting the execution code from the language preferably from 'C'.

join(): If the thread has to wait until the other thread finished its execution then we need to go for `join()`.

If `t1` execute `t2.join()` then `t1` should wait till `t2` finishes its execution.

`t1` will be entered into waiting state until `t2` completes, once `t2` completes then `t1` will continue its execution.

eg:: veneer fixing → `t1.start()`
 wedding Card printing → `t2.start()` → `t1.join()`
 wedding Card distribution → `t3.start()` → `t2.join()`

Prototype of join()

public final void join() throws InterruptedException

public final void join() throws InterruptedException
 (long ms)

public final void join (long ms, int ns) throws InterruptedException

Note: While one thread is in Waiting State and if one more thread interrupts then it would result in "InterruptedException". It's checked exception and should be handled.

Thread t = new Thread(); // born state

t.start(); // ready state

- * If T.S. allocates CPU time then Thread enters into running state.
- * If currently executing thread invokes t.join() / t.join(1000), t.join(1000, 10), then it would enter into waiting state.
- * If the thread finishes the execution / time expired / interrupted then it would come back to ready state / runnable state
- * If run() is completed then it would enter into dead state.

eg:: Class MyThread extends Thread {
 @Override

 public void run() {

 for (int i=1; i<5; i++) {

 System.out.println("Sira thread");

 try {

 Thread.sleep(2000);

 } catch (InterruptedException e) { }

 }

}

 public class Test3 {

 public static void main(String... args) throws

 InterruptedException {

 MyThread t = new MyThread();

 t.start();

 t.join(10000); // line n

 for (int i=1; i<5; i++) { System.out.println("rama thread"); }

}

}

If the line `n1` is commented then we can't predict the output because it is the duty of the T.S to assign CPU time.

If line `n1` is not commented, then main thread (main thread) will enter into waiting state, till sita thread (child thread) finishes its execution.

Output : 2 Thread : a. Child thread

Sita thread

Sita thread

b. Main thread

Rama thread

Rama thread

Waiting of child thread until Completing main thread:

We can make main thread to wait for child thread as well as we can make child thread also to wait for main thread.

eg:: Class Mythread extends Thread{

 Static Thread mt;

 @Override

 Public void run(){

 try{ mt.join(); }

 Catch (InterruptedException e){ }

 For (int i=0; i<10; i++) Sysout ("child thread"); }

}

Public class Test3{

 Public static void main (String [] args) throws InterruptedException

 { myThread . mt = new Mythread(); }

 Thread . currentThread () ;

```

MyThread t = new MyThread();
t.start();
for (int i=0; i<10; i++) {
    System.out.println("main-thread");
    Thread.sleep(2000);
}
}

```

Output: MainThread : main thread

Child Thread : child thread

eg::

```

Class MyThread extends Thread {
    static Thread mt;

```

@Override

```

public void run() {
    try {
        mt.join();
    }

```

Catch (InterruptedException e) { }

```

for (int i=0; i<10; i++) System.out.println("Child thread");
}

```

public class Test3 {

public static void main (String [] args) throws InterruptedException {

MyThread mt = Thread.currentThread();

MyThread t = new MyThread();

t.start();

t.join();

for (int i=0; i<10; i++) System.out.println("main thread");

Thread.sleep(2000); }

}

}

Output: 2-threads (Main, Child-thread)
 main-thread
 child-thread.

Note: If both the threads invoke `t.join()`, `mt.join()` then the program would result in "deadlock".

eg:: public class Test3 {

```
public static void main (String ...a) throws InterruptedException
{ Thread.currentThread().join(); }
```

}

Output: Deadlock!, because main thread is waiting for main thread itself.

Sleep(): If a thread don't want to perform any operation for a particular amount of time then we should go for `Sleep()`.

Signature:

- * `public static native void sleep (long ms)` throws `InterruptedException`
- * `public static void sleep (long ms, int ns)` throws `InterruptedException`

* Every `Sleep` method throws `InterruptedException`, which is a checked exception so we should compulsorily handle the exception using `try catch` or by `throws` keyword otherwise it would result in Compile Time Error.

Thread `t = new Thread();` // new or born state
`t.start();` // ready/runnable state

- * If T.S allocated CPU time then it would result/enter into running state.
- * If `run()` complete then it would enter into dead state.

- * If running thread invokes `sleep(1000)` / `sleep(1000, 100)` then it would enter into Sleeping State.
- * If time expired / if sleeping thread got interrupted then thread would come back to "ready / runnable State".

eg::

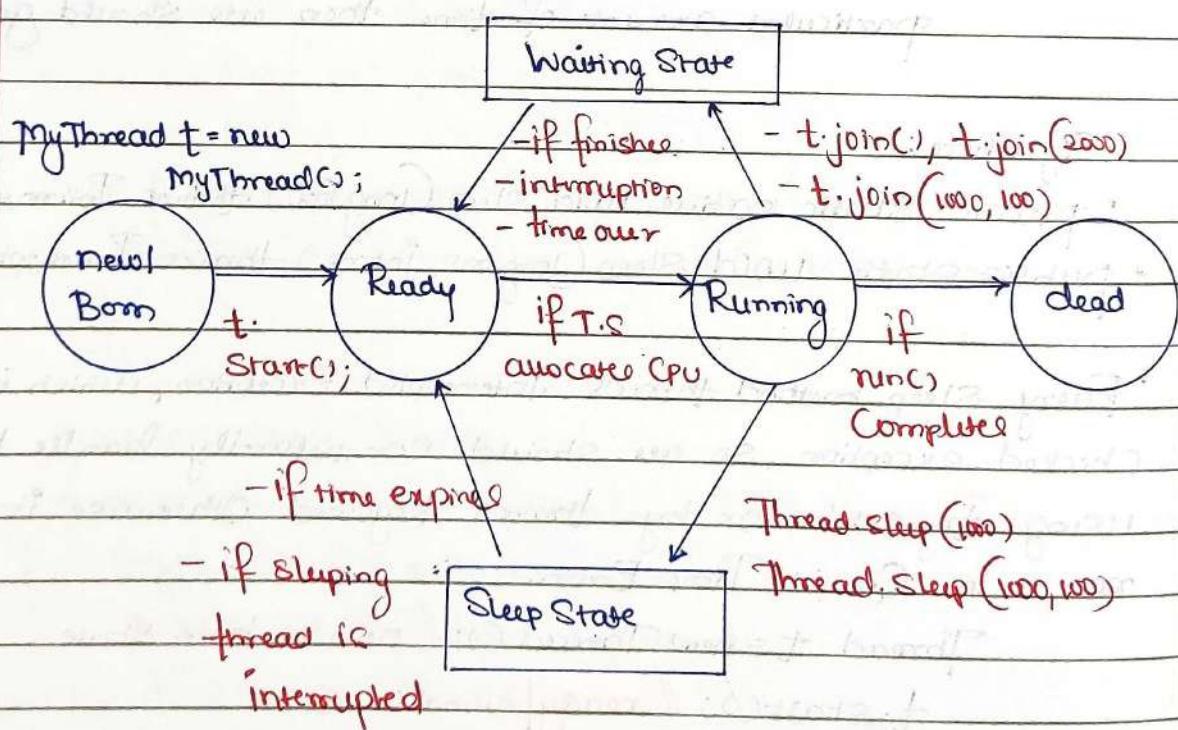
```

public class SlideRotator {
    public static void main(String[] args) {
        throw InterruptedException
        { for(int i=0; i<10; i++) {
            System.out.println("Slide: " + i);
            Thread.sleep(5000);
        }
    }
}

```

Output: Slide: 1
Slide: 2
.....

Life cycle of Thread



Interrupting a Thread

Public void interrupt()

- If thread is in sleeping state or in waiting state we can interrupt a thread.

eg:: Class MyThread extends Thread{

@Override

public void run(){
try{

for(int i=0; i<10; i++) {

Sayout("I am a lazy thread");

Thread.sleep(2000); }

}

Catch (InterruptedException e)

{ Sayout("I got interrupted"); }

}

public class Test{

public static void main (String [] args) throws InterruptedException{

{ MyThread t = new MyThread();

t.start();

t.interrupt();

Sayout("End of main"); }

}

Output: main thread : main thread

Child Thread: I am a lazy thread

I got interrupted

eg::

Class MyThread extends Thread{

@Override

public void run(){

Note: * If thread is interrupting another thread, but target thread is not in waiting state / sleeping state then there would be no exception.

- * `Interrupt()` call be waiting till target thread enters into waiting state/sleeping state so that this call won't be wasted.
 - * Once the target thread enters into waiting state/sleeping state then `Interrupt()` will interrupt and it cause the exception.
 - * `Interrupt` call will be wasted only if thread does not enter into waiting state/sleeping state.

yield() join() sleep()

1. Purpose:

- * **yield()** - To pause current executing Thread for giving the chance of remaining waiting Threads of same Priority.
- * **join()** - If a thread wants to wait until completing some other thread then we should go for join.
- * **Sleep()** - If a thread don't want to perform any operation for a particular amount of time then we should go for sleep.

2. Is it static?

yield() - yes
join() - no
sleep() - yes.

3. Is it final?

yield() - no
join() - yes
sleep() - no

4. Is it overloaded?

yield() - no
join() - yes
sleep() - yes

5. Throw IE?

yield() - no
join() - yes
sleep() - yes

6. Is it native method?

yield() - yes
join() - no
sleep():
 - **Sleep(long ms)** - native
 - **Sleep(long ms, int os)** - non-native

Note: Using Lambda expression:

```
Runnable r = () -> {
    for(int i=0; i<5; i++)
    { System.out.println("child thread"); }
};
```

```
Thread t = new Thread(r);
t.start();
```

Using anonymous inner class

```

new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i=5; i<10; i++) System.out.println("child-thread");
    }
}).start();

```

Synchronization:

1. Synchronized is a keyword applicable only for methods and blocks.
2. If we declare a method/block as synchronized then at a time only one thread can execute that block/method on that object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. But the main disadvantage of synchronized keyword is it increases waiting time of thread and affects performance of the system.
5. Hence if there is no specific requirement then never recommend to use synchronized keyword.
6. Internally synchronized concept is implemented by using lock concept.

Class X {

 synchronized void m1() {}

 synchronized void m2() {}

 void m3() {}

}

Key points:

- * If t₁ thread invokes m₁₍₎ then on the Object x lock will be applied
- * If t₂ thread invokes m₂₍₎, then m₂₍₎ can't be called because lock of x Object is with m₁₍₎.
- * If t₃ thread invokes m₃₍₎ - then execution will happen because m₃₍₎ is non Synchronized. Lock concept is applied at the Object level not at method level.

7. Every Object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into picture.
8. If a thread wants to execute any synchronized method on the given object, first it has to get the lock of that object. Once the thread got the lock of that object then it will allow to execute any synchronized method on that object. If the synchronized method on that object execution completed then automatically thread releases lock.
9. While a thread executing any synchronized method the remaining threads are not allowed to execute any synchronized method on that object simultaneously. But remaining threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method]

Note: Every object will have 2 area [Synchronized Area and Non Synchronized Area]

Synchronized area: Write a code only to perform update, insert, delete.

Non Synchronized area: Write the code only to perform select operation.

Class ReservationApp{

 Check Availability() { // performs read opn }

```
Synchronized BookTicket() {
    // perform update operation
}
```

eg:: Class Display {
 public void wish(String name)
 {
 for(int i=0; i<10; i++)
 {
 System.out("Good Morning:");
 try {
 Thread.sleep(2000);
 }
 catch(InterruptedException e) {
 System.out(name);
 }
 }
}

Class MyThread extends Thread {

Display d;

String name;

MyThread (Display d, String Name)

```
{  

    this.d = d;  

    this.name = name;
}
```

@Override

public void run()

d.wish(name);

}

Public class Test3 {

public static void main (String... args)

```
{  

    Display d = new Display();
}
```

MyThread t1 = new MyThread (d, "Dhoni");

MyThread t2 = new MyThread (d, "Sachin");

```

    t1.start();
    t2.start(); } }
```

Output: As noticed below the output is irregular because at a time on a resource called wish() 2 threads are simultaneously.

3 Threads:

Main Thread	Good Morning :	Good morning :
Child Thread-1	:	:	
Child Thread-2	:	:	

eg:: 2.

Class Display{

```

public synchronized void wish (String Name)
{ for (int i=0; i<10; i++)
  {
```

```

    System.out.print ("Good morning : ");
    try {
```

```

    Thread.sleep (2000); }
```

```

  catch (InterruptedException e) { } }
```

```

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

Class MyThread extends Thread {

Display d;

String name;

MyThread (Display d, String name)

```

{ this.d=d;
  this.name=name; }
```

@Override

```

public void run () { d.wish (name); }
```

}

```
public class Test3 {
```

```
    public static void main (String... args) throws InterruptedException {
        Display d = new Display();
        MyThread t1 = new MyThread(d, "dhoni");
        MyThread t2 = new MyThread(d, "yuvvi");
        t1.start();
        t2.start();
    }
}
```

Output: 3 threads:

a. Main Thread

b. Child Thread-1

Good Morning: dhoni

c. Child Thread-2

Good Morning: yuvvi

Note: As noticed above there are 2 threads which are trying to operate on single object called "display" we need "Synchronization" to resolve the problem of "Data Inconsistency".

Case Study:: Display d1 = new Display();

Display d2 = new Display();

MyThread t1 = new MyThread(d1, "yuvraj");

MyThread t2 = new MyThread(d2, "Sachin");

t1.start();

t2.start();

In the above case we get irregular output, because two different object and since the method is synchronized lock is applied w.r.t object and both the threads will

Start simultaneously on different java objects due to which the output is "irregular".

Conclusion:

- * If multiple threads are operating on multiple objects then there is no impact of Synchronization.
- * If multiple threads are operating on same java objects then Synchronized Concept is Applicable.

Class level lock

1. Every class in java has a unique level lock.
2. If a thread wants to execute static synchronized method then the thread requires "Class level lock".
3. While a thread executing any static synchronized method the remaining threads are not allowed to execute any static synchronized method of that class simultaneously.
4. But remaining threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.
5. Class level lock and object lock both are different and there is no relationship between these two.

eg:: Class X {

 Static synchronized m1() {} // Class level lock

 Static synchronized m2() {}

 Static m3() {} // no lock required

 Synchronized m4() {} // Object level lock

 m5() {} // no lock required.

}

t₁ — m1() → Class level lock is applied and chance is given

t₂ — m2() → enters into waiting state

t3 - m3() → gets a chance of execution without any lock
 t4 - m4() → Object level lock applied and chance is given.
 t5 - m5() → gets a chance for execution without any lock.

eg:: 1

```

Class Display {
  public synchronized void displayNumbers () {
    for (int i=1; i<10; i++) {
      System.out.println(i);
      try { Thread.sleep(2000); }
      catch (InterruptedException e) { }
    }
  }

  public synchronized void displayCharacters () {
    for (int i=65; i<=75; i++) {
      System.out.println((char)i);
      try { Thread.sleep(2000); }
      catch (InterruptedException e) { }
    }
  }
}
  
```

Class MyThread1 extends Thread {

Display d;

MyThread1 (Display d) { this.d=d; }

@Override

public void run () { d.displayNumbers (); }

Class MyThread2 extends Thread {

Display d;

MyThread2 (Display d) { this.d=d; }

@Override

public void run() { d.displayCharacters(); }

}

public class Test3 {

public static void main (String ... args)

{ Display d1 = new Display();

MyThread t1 = new MyThread (d1);

MyThread t2 = new MyThread (d1);

t1.start();

t2.start(); }

}

Output: 3 Thread

a. Main Thread

b. User Defined Thread

displayCharacters()

c. User Defined Thread

displayNumbers()

Synchronized block

Synchronized void m1() {

: : :

}

* If few lines of code is required to get synchronized then

it is not recommended to make method only as synchronized.

* If we do this then for threads performance will be low, to
resolve this problem we use "Synchronized block", due to which
performance will be improved.

Case Study:

- a) If a thread got a lock of current object, then it is allowed to execute that block a.

Synchronized (this){

⋮⋮⋮

}

- b) To get a lock of particular Object :: B.

Synchronized (B){

⋮⋮⋮

}

* If a thread got a lock of particular Object B, then it is allowed to execute that block.

- c) To get Class level lock we have to declare Synchronized block as follows:

Synchronized (Display.class){

⋮⋮⋮

}

* If a thread gets class level lock, then it is allowed to execute that block.

Synchronized block (continued)

eg::1 Class Display{

public void wish (String name){

⋮⋮⋮ // many lines

Synchronized (this){

for (int i=0; i<5; i++)

{ Snow ("Good Morning");

try{ Thread.sleep (2000); }

Catch (InterruptedException e){ }

```
Sayout(name); }
```

```
}
```

}; // many lines of code

```
Class MyThread extends Thread {
```

```
Display d;
```

```
String name;
```

```
MyThread (Display d, String name)
```

```
{ this.d = d;
```

```
-this.name = name; }
```

```
Public void run() { d.wish(name); }
```

```
}
```

```
Public static void main (String... args)
```

```
{ Display d = new Display();
```

```
MyThread t1 = new MyThread (d, "dhan");
```

```
MyThread t2 = new MyThread (d, "yuv");
```

```
-t1.start();
```

```
-t2.start(); }
```

Output:
Good Morning yuv
Good Morning yuv
:
:
:

eg:: 2 (for the above code)

```
Public static void main (String... args)
```

```
{ Display d1 = new Display();
```

```
Display d2 = new Display();
```

```
MyThread t1 = new MyThread (d1, "dhan");
```

```
MyThread t2 = new MyThread (d2, "yuv");
```

```
t1.start();
```

```
t2.start(); }
```

Output:
Inreguar

eg:: 3 (for same code in display class)

Synchronized (Display class)

{ ... }

public static void main (String ... args)

{ Display d1 = new Display();

Display d2 = new Display();

MyThread t1 = new MyThread (d1, "dhoni");

MyThread t2 = new MyThread (d2, "yuvil");

t1.start();

t2.start(); }

Note: * 2 object, 2 thread, but the thread which gets a chance applied class level lock so output is regular.

* Lock concept applicable only for objects and class types, but not for primitive types, if we try to do it, it would result in compile time error saying "unexpected type".

eg:: int x = 10;

synchronized (x) { .. } // CE: unexpected type.

... }

Inter Thread Communication (remember postbox example)

Two threads can communicate with each other with the help of

- a. notify()
- b. notifyAll()
- c. wait()

notify(): Thread which is performing updation should call notify() so waiting thread will get notification so it will continue with its execution with the updated items.

Wait(): Thread which is expecting notification / updation should call `wait()`, immediately the Thread will enter into waiting State.

- * If a thread wants to call `wait()`, `notify()` / `notifyAll()` then Compulsorily the thread Should be the owner of the object otherwise it would result in "IllegalMonitorStateException".
- * We say thread to be the owner of that Object if thread has lock of that Object.
- * It means these methods are part of synchronized block or method, if we try to use outside Synchronized Area then it would result in Runtime Exception Called "IllegalMonitorStateException".
- * If a thread calls `wait` on any object, then first it immediately releases the lock on that object and it enters into waiting State.
- * If a thread calls `notify` on any object, then he may or may not release the lock on that object immediately.
- * Except `wait()`, `notify()`, `notifyAll()`, lock can't be released by other methods.

Note:

- * `yield()`, `Sleep()`, `join()` - Can't release the lock
- * `Wait()`, `notify()`, `notifyAll()` - will release the lock, Otherwise interthread communication Can't happen.
- * Once a Thread calls `wait()`, `notify()`, `notifyAll()` methods on any Object then it releases the lock of that particular object but not all locks it has.

Method prototype:

1. `public final void wait() throws InterruptedException`
2. `public final native void wait(long ms) throws InterruptedException`
3. `public final void wait(long ms, int ns) throws InterruptedException`

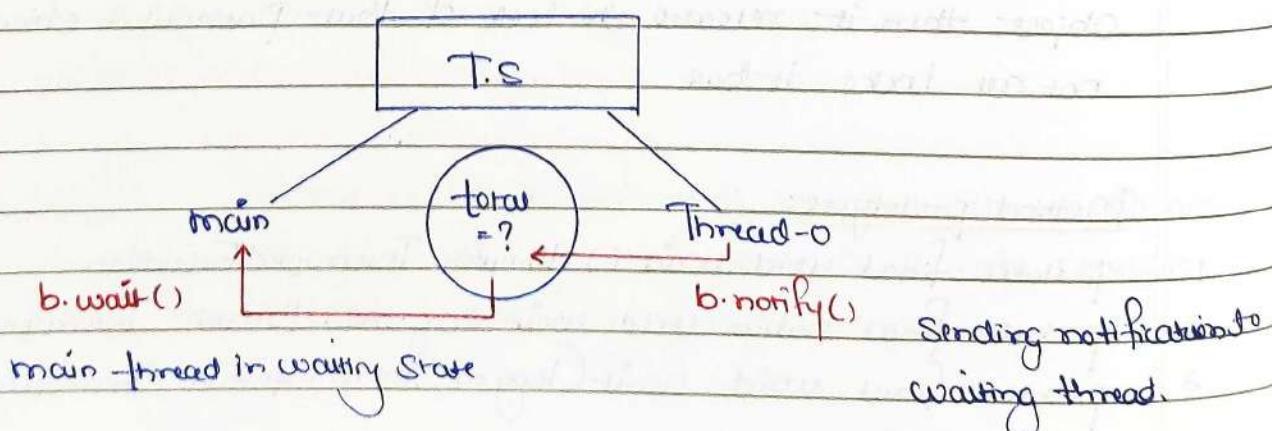
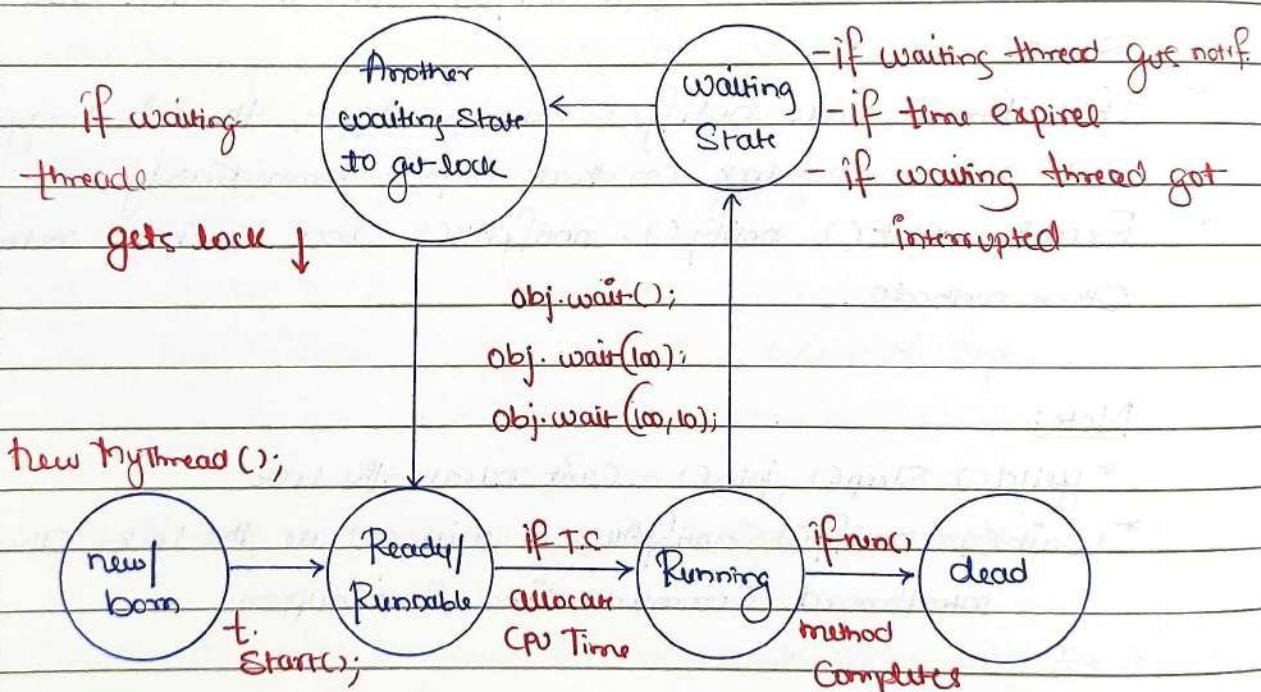
4. public final native void notify()
 5. public final void notifyAll()

Interview Questions.

Q1. Methods like wait(), notify(), notifyAll() are present inside Object Class, why not in Thread Class?

Ane. Thread will call wait(), notify(), notifyAll() on Objects like PostBox, Stack, Customer...
 → Obj.wait(), Obj.notify(), Obj.notifyAll()

These methods should be available for every object in java, if method had to be available for every object in java then those methods should come from "Object" class.



- * main thread got the notifications, waited for the lock to be released from the other thread, once the lock is released it will use ThreadB total variable

eg:: class ThreadB extends Thread{

 int total=0;

 @Override

 public void run(){

 for (int i=0; i<100; i++)

 { total+=i; }

}

 public static void main (String ... args)

 { Thread B b=new ThreadB();

 b.start();

 // Statement

 System.out.println(b.total); }

A) If we replace the "Statement" with `Thread.sleep(10000)` then thread will enter into waiting state but within 10s the update value is ready. Within 10s if the update is not ready, then we should not use `Thread.sleep(10000)`;

B) If we replace with `b.join()`, then main thread will enter into waiting state, then child will execute for loop, till the main thread has to wait.

main thread is waiting for update result.

`for (int i=0; i<100; i++) { total+=i; }`

// 1 Cr line of code is available

main thread has to wait till 1Cr line of code, why main thread should wait for the completion of the code.

eg:: Class ThreadB extends Thread {

```
    int total = 0;
```

@Overide

```
public void run() {
```

```
    synchronized (this) {
```

```
        System.out.println("child thread started");
```

```
        for (int i = 0; i <= 100; i++) total += i;
```

```
        System.out.println("child thread giving notification");
```

```
        this.notify();
```

```
}
```

("different class")

public static void main (String [] args) throws InterruptedException

```
{ ThreadB b = new ThreadB();
```

```
    b.start();
```

```
    Thread.sleep(10000);
```

```
    synchronized (b) {
```

```
        System.out.println("main thread is calling wait on object B");
```

```
        b.wait();
```

```
        System.out.println("Main thread got notification");
```

```
        System.out.println(b.total);
```

```
}
```

Output: Child thread Started Calculation

Child thread trying to give notification

Main thread is calling wait on object B

because of Thread.sleep(10000) main thread will need
get notification.

eg.: (for same code)

```
public static void main (String [] args) throws InterruptedException
{
    Thread B b = new Thread (B ());
    b.start ();
```

Synchronized (b){

```
System.out ("main thread calling wait on B object");
b.wait (10000);
System.out ("main thread got notification");
System.out (b.total); }
```

}

Output: Child thread started Calculation

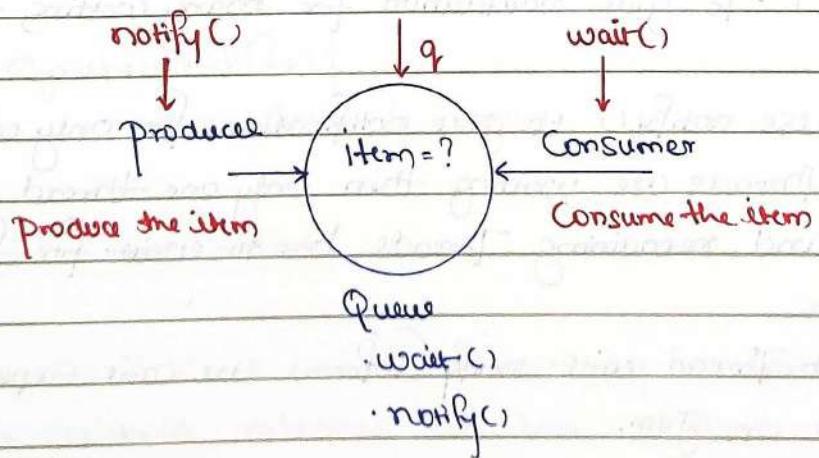
Child thread trying to give notification

Main thread is waiting on B Object

Main thread got notification

5050

Producer and Consumer Problem:



Producer: produce the item and update in the queue.

Consumer: consume the item from the queue.

Class Producer extends Thread {

Producer () {

 Synchronized (q) {

 // produce items and update to queue

 q.notify(); }

}

Class Consumer extends Thread {

Consumer () {

 Synchronized (q) {

 if (q.isEmpty) { q.wait(); }

 else

 // consume item from the Queue }

}

Difference between notify() and notifyAll()

- * notify() → To give notification for one waiting thread.
- * notifyAll() → To give notification for many waiting threads.

→ We can use notify() to give notification for only one thread. If multiple threads are waiting then only one thread will get a chance and remaining threads have to wait for further notifications.

But which thread will notify (inform) we can't expect because it depends on Jum.

Waiting State

Obj. notify()

Obj1.wait(); // Go threads waiting.
Obj1.notify();

Running State

Among 60 threads which thread will get a chance we don't have control over that, it is decided by Jvm (Thread Scheduler).

→ We can use notifyAll() method to give notification for all waiting threads of particular object.

All waiting threads will be notified and will be executed one by one, because they required lock.

<u>Waiting State</u>	<u>Note:</u> On which Object we are calling wait(), notify(), and notifyAll() that corresponds to object lock we have to get but not other object locks.
Obj.wait()	Obj.wait();
Obj.notifyAll()	Obj2.wait();

Running State

e.g.: Stack s1 = new Stack();

Stack s2 = new Stack();

Synchronized (s1){

s2.wait(); } // RE: Illegal Monitor State Exception

Synchronized (s2){

s2.wait(); } // valid

Questions based on lock

- If a thread calls wait() immediately it will enter into waiting state without releasing any lock → False
- If a thread calls wait() it releases the lock of that object but may not immediately → False.
- If a thread calls wait() on any object, it releases all locks

acquired by that thread and enters into waiting state \rightarrow False.

4. If a thread calls `wait()` on any object, it immediately releases the lock of that particular object and enters into waiting state \rightarrow True
5. If a thread calls `notify()` on any object, it immediately releases the lock of that particular object. \rightarrow Invalid
6. If a thread calls `notify()` on any object, it releases the lock of that object but may not immediately \rightarrow True.

Deadlock

- * If two threads are waiting for each other forever (without end), such type of situation (infinite waiting) is called deadlock.
- * There is no resolution technique for deadlock but several prevention (avoidance) techniques are possible.
- * Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.

eg:: Class A {

```
public void d1(B b){  
    try{ Thread.sleep(5000); }  
    catch(InterruptedException e){}  
}
```

System.out("Thread-1 trying to call b.last");

```
b.last(); }
```

public void last(){

```
System.out("Inside A.last()"); }
```

}

Class B {

```
public void d2(A a){
```

```
System.out("Thread-2 starts execution of d2"); }
```

```

try { Thread.sleep(5000); }
catch (InterruptedException e) { }
System.out.println("Thread-2 - trying to call A last()");
a.last();}

public void last() { System.out.println("Inside B last() method"); }
}

```

Public class Test extends Thread {

A a = new A();

B b = new B();

```

public void run() { this.start();
a.d1(b); } // executed by main thread

```

```

public void run() { b.d2(a); } // executed by child thread

```

public static void main (String... args)

{ Test t = new Test();

t.run(); }

Since methods are not synchronized, lock is not required, so no deadlock.

Output: Thread-1 Starts execution of d1()

Thread-2 Starts execution of d2()

Thread-1 trying to call B last()

Inside B last() method

Thread-2 trying to call A last()

Inside A last() method

eg:: 2 (using Synchronized keyword for every method in A, B)

public static void main (String... args)

(Run-on code)

{ Test t = new Test();

t.run(); } // main thread executing

Same as above)

In the above program, there is a possibility of "deadlock".

Output: Thread-1 starts execution of d1()

Thread-2 starts execution of d2()

Thread-1 trying to call B.laet()

Thread-2 trying to call A.laet()

// here cursor will be waiting.

→ t1 → Starts d1(), Since d1() is synchronized and a part of 'A' class so t1 applied lockof(A) and starts the execution, while executing it encounters Thread.Sleep(). So TS gives chance for t2 thread.

After getting a chance again by TS, it tries to execute b.laet but after lock of b is with t2 thread, so t1 enters into Waiting State.

→ t2 → Starts d2(), Since d2() is synchronized and a part of 'B' class to t2 applied lockof(B) and starts the execution, while executing it encounters Thread.Sleep(), So TS gives chance again for t1 thread.

After getting a chance again by T2, it tries to execute a.laet but lock of a is with t1 thread, so t2 enters into Waiting State.

Since both threads are in waiting state and it would be waiting forever, So we say the above program would result in "Deadlock".

Note: Synchronized is the only reason why there is a deadlock, so we should be careful when we use synchronized keyword, if we remove atleast one synchronized word - then the program would enter into "deadlock".

Deadlock vs Starvation

Long waiting of a thread, where waiting never ends is termed "deadlock".

Long waiting of a thread, where waiting ends at a certain point is called "Starvation".

eg:: Assume we have 1 or threads, where all 1 or threads have priority of 10, but one thread is of priority 1 and it has to wait for long time, this scenario is called "Starvation".

Note: Low priority thread has to wait until completing all priority threads but ends at certain point which is nothing but starvation.

Daemon Thread: The thread which is executing in background is called "Daemon Thread". eg: Attach Listener, Signal Dispatcher ...

Main Objective of Daemon Thread:

* To provide support for Non-Daemon threads (main thread).

eg:: if main thread runs with low memory then jvm will call Garbage Collector thread to cleanup the useless objects, so that no. of bytes of free memory will be improved, with this free memory main thread can continue its execution.

Usually Daemon threads have low priority, but based on our requirement Daemon threads can run with high priority also.

JVM → Create 2 threads

a. Daemon thread (priority 1, Priority 10)

b. main (Priority = 5)

While executing the main code, if there is a shortage of memory then immediately JVM will change the priority of Daemon thread to 10, so Garbage Collector activates Daemon

thread and it frees the memory after doing it immediately it changes the priority to 1, so main thread will continue.

Q1 How to check whether the Thread is Daemon or not?

Ane * public boolean isDaemon() → To check whether the thread is Daemon.

* public boolean setDaemon(boolean b) throws IllegalThreadStateException

b → true, means the thread will become Daemon, before starting the thread we need to make the thread as "Daemon" otherwise it would result in "IllegalThreadStateException".

Q2 What is the default nature of the Thread?

Ane By default the main thread is "NonDaemon", for all remaining thread Daemon nature is inherited from Parent to Child, that is if the parent thread is "Daemon" then child thread is also "NonDaemon".

Q3 Is it possible to change the NonDaemon nature of main thread?

Ane No, because the main thread's starting is not in our hand, it will be started by JVM.

eg:: class MyThread extends Thread { }

public class Test { }

public static void main (String... args) { }

System.out.println(Thread.currentThread().isDaemon()); // false

Thread.currentThread().setDaemon(true); // RE

MyThread t = new MyThread();

System.out.println(t.isDaemon()); // false

t.setDaemon(true);

```
t.start();
syout(t.isDaemon); } // true
}
```

Note: Whenever last daemon threads terminate, automatically all daemon threads will be terminated irrespective of their position.

eg:- makeup man is shooting is a daemon thread
 - hero is main thread
 - if hero role is over, then automatically the makeup role is also over.

```
eg:: class MyThread extends Thread {
  public void run() {
    for(int i=1; i<=10; i++) {
      syout("child thread");
      try { Thread.sleep(200); }
      catch(InterruptedException e) { syout(e); }
    }
  }
}
```

```
public class Test {
  public static void main(String... args) {
    MyThread t = new MyThread();
    t.setDaemon(true); // Stmt - 1
    t.start(); syout("end of main"); }
}
```

Output: if we comment Stmt-1,
 both threads are non daemon
 threads, so it would continue executing.
 end of main-thread

Child thread

Output: in the above code, main
 thread is non daemon and user defined
 thread is daemon, if main thread
 finished execution, automatically
 daemon thread will also finish the
 execution.

Collections in Java

The main 7 classes in collections are

1. ArrayList
2. LinkedList
3. ArrayDeque
4. PriorityQueue
5. TreeSet
6. HashSet
7. LinkedHashSet

Divided into 3 major interfaces:

1. List
2. Queue
3. Set

① ArrayList → List(I)

- * Internally dynamic array data structure
- * Size of the ArrayList grows automatically
- * Heterogeneous data elements can be stored with homogeneous
- * Index based accessing allowed
- * We are able to add elements at any given index

Syntax: `ArrayList al1 = new ArrayList();`
`al1.add(30);`
`al1.add(20);`
`System.out.println(al1); // [30, 20]`

```
ArrayList al2 = new ArrayList();
al2.add("vinay");
al2.add(45.9);
System.out.println(al2); // ["vinay", 45.9]
```

```
al3.add(ss);
// add at rear
al3.add(0, 7);
// add at first
al3.add(7, 9);
```

```
ArrayList al3 = new ArrayList();
al3.add(0); // Can add one
```

Complete
arraylist.

But adding at index is
inefficient

2) LinkedList → List(I) and Deque(I)

- * Doubly linked list data structure.
- * Homogeneous data can be stored, duplicate are allowed.
- * Heterogeneous data can be stored.
- * Data is stored as an object, index based accessing allowed.

Syntax: `LinkedList l1 = new LinkedList();`
`l1.add(s0);`
`l1.add("ineuron");`
`System.out.println(l1); // [s0, ineuron]`

`l1.addFirst("Hyderabad"); // add at first`
`l1.add(3, qq); // adding at index 3`
`l1.addLast("Bangalore"); // add at last`

ArrayList and LinkedList have almost same methods but LinkedList has some extra methods and adding at particular index is very fast compared to ArrayList because of doubly LL data structure

Q.1. When to use Array over ArrayList?

Ans Whenever the size of the data is known and you are sure that the data is homogeneous then you must go with array.

Array is faster than ArrayList because in ArrayList data is stored as an Object which is little time consuming, for primitive data types also get converted to objects, but array store as it is.

Other methods in ArrayList

- * `a14.get(5); // used to fetch element from position 5.`
- * `a14.contains(44); // returns true if 44 is available`
- * `a14.indexOf(22); // returns index of 22`
- * `a14.isEmpty(); // checks if empty`
- * `a14.size(); // returns number of elements`
- * `a14.trimToSize(); // clear the extra space`
- * `a14.clear(); // erases all data from ArrayList.
(and many more are available)`

Other methods in Linked List

- * list.clear(); // removes all elements from list.
- * list.getFirst(); // gives you first item
- * list.getLast(); // gives you the last item
- * list.indexOf(40); // returns index of 40
- * list.lastIndexOf(40); // returns last index of 40
- * list.addFirst(50); // adds element at first
- * list.addLast(90); // adds element at last
- * list.peekFirst(); // returns first item
- * list.popFirst(); // gives first item, and item is removed from list.
(and many more available)

③ ArrayDeque → Deque(I)

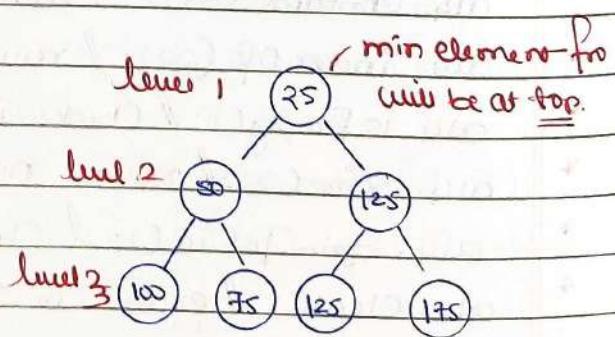
- * Double ended queue data structure.
- * Index based accessing is not allowed.
- * Insertion and deletion at front and back are allowed but not in middle.
- * Duplicate allowed.

Syntax : ArrayDeque ad = new ArrayDeque();
ad.add(10); // adds element
ad.add(10);
ad.addFirst(30); // adds at first
ad.addLast(70); // adds at last.

④ PriorityQueue → Queue(I)

- * Min heap data structure
- * Indexing is not allowed
- * Duplicates are allowed
- * Element will be stored

in form of min heap data



Structure - higher priority element will be at the top.

(5) TreeSet → Set(I)

- * Binary Search tree data structure.

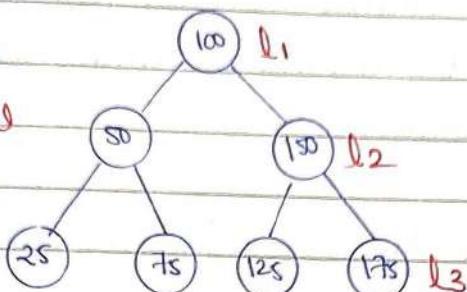
{ 100, 50, 150, 25, 75, 125, 175 }

in-order

traversal

for

Search



* Duplicate not allowed.

* If an element is less than root, go to left, if higher than right.

* Searching for key(20) takes 3 Comparisons in this example.
i.e. $O(\log n)$ time complexity.

- * When the data is in sorted order, it forms a "squared tree" and searching takes 'n' comparisons i.e. $O(n)$ TC

Syntax: `TreeSet ts = new TreeSet();`

`ts.add(50); // add element`

`ts.add(75);`

`ts.add(50);`

`ts.ceiling(50); // returns 50 if available, or next higher value`

`ts.higher(50); // returns next higher element than 50.`

`ts.floor(40); // returns 40 if available, or next lower element`

`ts.floor(40); // "`

`ts.lower(40); // returns the first lower element than 40`

(6)

HashSet → Set(I)

- * It internally has a hash function (hashing algorithm) and associated HashTable with load factor(75%).

* Duplicate values are not allowed

* On an average the `contains()` of HashSet runs in $O(1)$ time.

* Order of insertion is not maintained

Syntax: `HashSet hs = new HashSet();`

`hs.add(100);`

7

LinkedHashSet

- * It is a sub class (child class) of HashSet.
- * It also follows hashing algorithm behind the scene to store data because of which searching becomes very fast.
- * The order of insertion is maintained unlike HashSet.
- * Duplicates are not allowed.

Syntax : LinkedHashSet lhe = new LinkedHashSet();
 lhe.add(10);
 lhe.add(50);

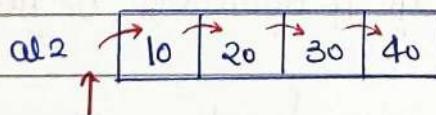
→ Different ways to access data from collection

1> `for (Object obj : al2) // only for classes that allow index
 { System.out.println(obj + " "); } based accessing (normal for).
 // for-each is for all classes.`

* Iterator itr1 = al2.iterator();

// Iterator is common to all classes in collection

iterator points to the first element (just before it)



* "itr1.next()" gives the next object value from the array list.

2> `while (itr1.hasNext()) // hasNext() returns boolean value
 { System.out.println(itr1.next()); } if there is an item next.`

* ListIterator litr = al2.listIterator(al2.size());

// ListIterator is applicable only for ArrayList and Linked List

3>

```
while (ditr.hasPrevious())
{ System.out.println(ditr.previous()); } // traverse in reverse direction.
using list iterator.
```



Iterator dirr = list.descendingIterator();

// descending iterator is only in Linked list, ArrayDeque,
TreeSet.

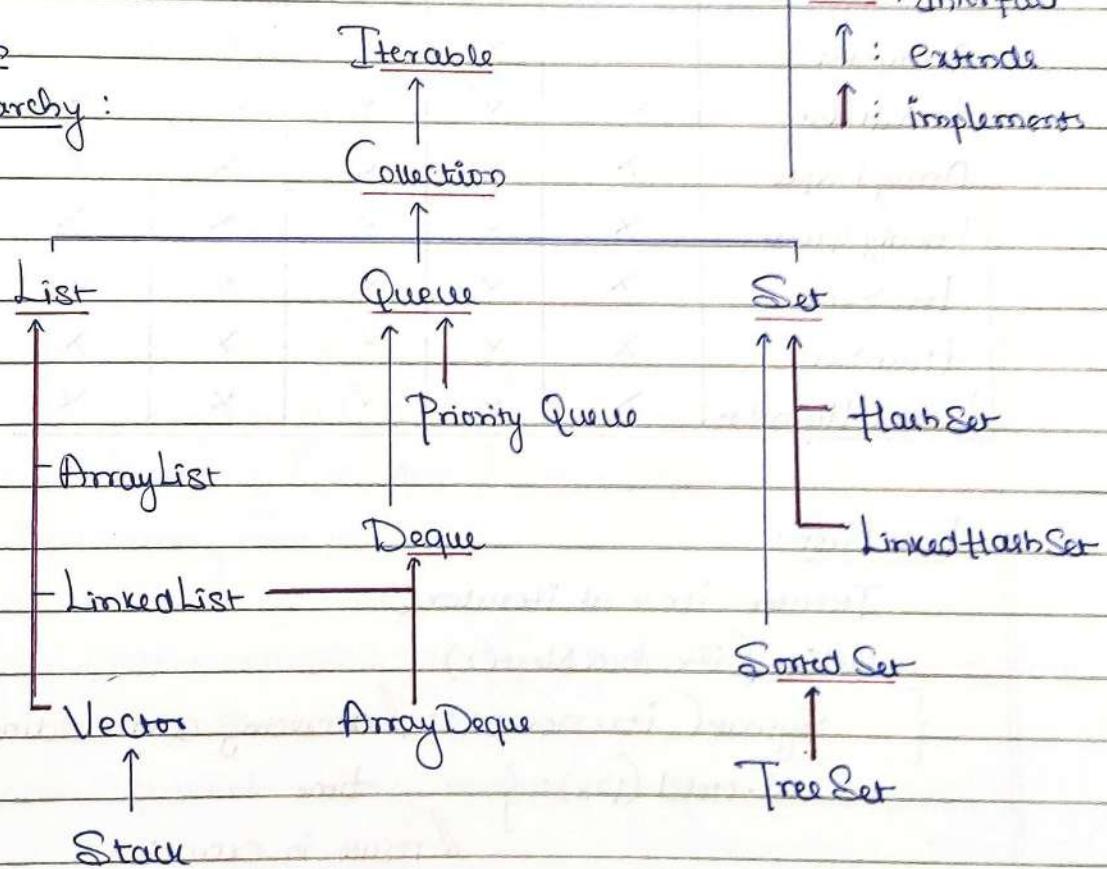
// Points to last item and when next() is called goes to
previous next (next in reverse order)

4>

```
while (ditr.hasNext());
{ System.out.println(ditr.next()); } // traversing in reverse order
```

Collection

Hierarchy:



Summary

Class	Internal DS	Presence order of insert.	Allow null value.	Duplicate allowed
ArrayList	Dynamic array	✓	✓	✓
LinkedList	Doubly linkedlist	✓	✓	✓
Array Deque	Double ended queue	✓	✗	✓
Priority Queue	min-heap ds	✗	✗	✓
TreeSet	binary search tree	✗	✗	✗
HashSet	hash table	✗	✓	✗
LinkedHashSet	hash table	✓	✓	✗

Class	for loop()	for each	Iterator	List Iterator	descending Iterator	Enumeration
ArrayList	✓	✓	✓	✓	✗	✗
LinkedList	✓	✓	✓	✓	✓	✗
Array Deque	✗	✓	✓	✗	✓	✗
Priority Queue	✗	✓	✓	✗	✗	✗
TreeSet	✗	✓	✓	✗	✓	✗
HashSet	✗	✓	✓	✗	✗	✗
LinkedHashSet	✗	✓	✓	✗	✗	✗

Fail Fast:

```

Iterator itr = al.iterator();
while (itr.hasNext())
{
    System.out.println(itr.next()); // accessing and adding at the same
    al.add(123); } // time
// result in exception
  
```

Fail Safe:

```

CopyOnWriteArrayList Col = new CopyOnWriteArrayList();
Col.add(100);
  
```

```
cal.add(2000);
```

```
cal.add(3000);
```

```
cal.add(4000);
```

```
Iterator itr = cal.iterator();
```

```
while (itr.hasNext())
```

```
{ System.out.println(itr.next()); // no exception, fair safe  
cal.add(12345); }
```

* Fair Safe means whenever we are attempting concurrent or structural modifications while accessing the data then it results in exception and Fair Safe.

* If you are attempting concurrent modifications, your modification should fail without exception then the concept of fair safe comes into picture.

* You can perform Fair Safe on classes which are available in Concurrent package (subpackage of util).

Inbuilt methods in Collections.

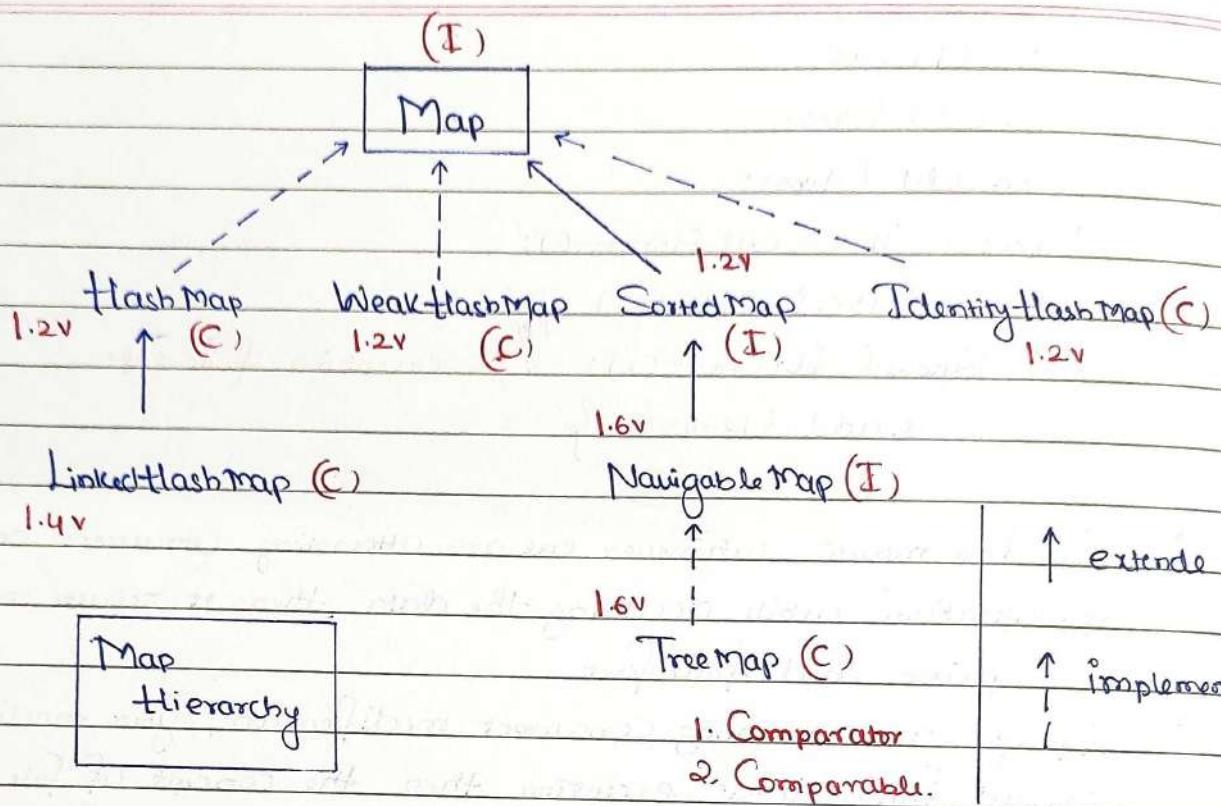
1. Collections.sort(cal); // sorts only if data is homogeneous
2. Collections.binarySearch(cal, 40); // returns index of 40 in array list
3. Collections.frequency(cal, 40); // returns frequency of 40 in cal
(and many more).

* All classes under collection API / framework have its own speciality (features, internal data structures).

* On objects present using method we perform actions.

Map in Java

The map interface is present in java.util package represents a mapping between a key and a value. The map interface is not a subtype of Collection Interface.



- * Maps are perfect to use for key-value associations mapping such as dictionaries.
 - * Since Map is an interface, Objects cannot be created of type map. We always need a class that extends this map.
 - * A map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap, LinkedHashMap but some do not like the TreeMap.
 - * The order of map depends upon implementations. TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
 - * There are two interfaces for implementing Map in Java: they are Map and SortedMap, and three classes: HashMap, TreeMap and LinkedHashMap.

Syntax : `HashMap hm = new HashMap();
hm.put(10, "Sachin");
hm.put(7, "MSD");`

```
hm.put(18, "Kohli");
```

```
System.out.println(hm); // 18 = Kohli, 7 = MSO, 10 = Sachin
```

// Order of insertion not maintained

```
LinkedHashMap hm = new LinkedHashMap();
```

```
hm.put(10, "Sachin");
```

```
hm.put(7, "MSO");
```

```
hm.put(18, "Kohli");
```

```
System.out.println(hm); // { 10 = Sachin, 7 = MSO, 18 = Kohli }
```

// Order of insertion is maintained

```
Collection c = hm.values();
```

```
Iterator itr = c.iterator();
```

```
while (itr.hasNext())
```

```
{ System.out.println(itr.next()); } // Kohli MSO Sachin
```

```
System.out.println(hm.get(10)); // Sachin
```

```
Set s = hm.keySet();
```

```
Iterator itr2 = s.iterator();
```

```
while (itr2.hasNext())
```

```
{ System.out.println(itr2.next()); } // 18 7 10
```

```
Set es = hm.entrySet();
```

```
Iterator itr3 = es.iterator();
```

// 18 = Kohli

```
while (itr3.hasNext())
```

7 = MSO

```
{ System.out.println(itr3.next()); } // 10 = Sachin
```

```
while (itr3.hasNext())
```

```
{ Map.Entry data = (Entry) itr3.next(); } // getting
```

```
System.out.println(data.getKey() + ":" + data.getValue()); } entries
```

Passport code to show Map functions:

```
import java.util.*;  
import java.util.Map.Entry;
```

Class Information

```
{ private String name;
```

```
private int age;
```

```
private String fatherName;
```

```
private String city;
```

Public Information (String name, int age, String fatherName,
String city)

```
{ this.name = name;
```

```
this.age = age;
```

```
this.fatherName = fatherName;
```

```
this.city = city; }
```

```
Public String getName() { return name; }
```

```
Public int getAge() { return age; }
```

```
Public String getFatherName() { return fatherName; }
```

```
Public String getCity() { return city; }
```

@Override

```
Public String toString()
```

```
{ return name + " " + age + " " + fatherName + " " + city; }
```

}

// System.out.print calls toString() method internally
so when we override it we get the desired
output instead of bare code.

```
public class LaunchPassport
{
```

```
    public static void main (String [ ] args)
    {
        Information info1 = new Information ("Rohan", 18, "Sharmaji",
        "Delhi");
    }
```

```
Information info2 = new Information ("Nitin", 27, "M", "Bengaluru");
```

```
Information info3 = new Information ("Hyder", 28, "H", "Bengaluru");
```

```
HashMap hm = new HashMap();
```

```
hm.put (000, info1);
```

```
hm.put (111, info2);
```

```
hm.put (333, info3);
```

```
Set set = hm.entrySet();
```

```
Iterator itr = set.iterator();
```

```
while (itr.hasNext())
```

```
{ Map.Entry passport = (Entry) itr.next();
```

```
System.out ("Passport Number : " + passport.getKey () + " Info : "
+ passport.getValue()); }
```

```
}
```

```
}
```

Output:

Passport Number : 0 : Info: Rohan 18 Sharmaji Delhi

Passport Number : 333 : Info Hyder 28 H Bengaluru

Passport Number : 111 : Info Nitin 27 M Bengaluru.

HashMap

LinkedHashMap

1. Came in 1.2v

2. DSA: HashTable

3. Order of insertion not preserved

4. Parent of LinkedHashMap

Came in 1.4v

DSA: HashTable + Linked List

Order of insertion maintained

Sub Class of HashMap

eg1.

```

import java.util.*;
class Employee
{
    private int getEmpid;
    private String name;
    private String empAddr;
    @Override
    public String toString() { return "Hyder"; }

    @Override
    public void finalize() {
        System.out("Garbage Collector Collected the object");
    }
}
public class LaunchGC
{
    public static void main(String[] args) throws Exception
    {
        Employee e = new Employee();
        e = null; // eligible for Garbage Collection
        System.gc(); // Call for garbage collector
    }
}

```

Output: Garbage Collector Collected the object

eg2.

```

public static void main(String[] args)
{
    Employee e = new Employee();
    HashMap hm = new HashMap();
    hm.put(e, "Hyder");
    e = null; // eligible for Garbage Collection
    System.gc(); // Call to GC but won't execute finalize
}

```

because hashmap dominates Garbage Collector.

eg:-

```

public static void main (String [ ] args)
{
    Employee e = new Employee ();
    WeakHashMap hm = new WeakHashMap ();
    hm.put (e, "Hyder");
    e = null; // eligible for gc
    System.gc(); // Call for GC, Since GC "dominates WeakHashMap"
    // we get the output from finalize block
}
* This is the only diff b/w HashMap and WeakHashMap
  
```

HashTable Syntax:

```

-HashTable ht = new HashTable ();
ht.put ("", "Hyder");
ht.put (12, "Nitin");
System.out.println (" { 11=Hyder, 12=Nitin } ");
  
```

HashMap	HashTable
<ol style="list-style-type: none"> 1. All methods are not synchronized 2. At a time multiple threads can operate on object. So not ThreadSafe. 3. Performance is high 4. Null is allowed for both keys and values (keys only once) 5. Introduced in 1.2v 	<ol style="list-style-type: none"> 1. All methods are synchronized 2. At a time Only One thread can operate on an object, so it is ThreadSafe. 3. Performance is low 4. Null is not allowed for both, results in NullPointerException. 5. Introduced in 1.0v

TreeMap Syntax:

```

TreeMap tm = new TreeMap ();
tm.put (14, "Rohan");
tm.put (12, "Vivek");
System.out.println (" { 12=Vivek, 14=Rohan } ");
  
```

Map Keypoints:

- * It is not a child interface of collection.
- * If we want to represent group of objects as key-value pair then we need to go for map.
- * Both Key and Value are Objects Only
- * Duplicate keys are not allowed but values are allowed.
- * Key-Value pair is called as "Entry".

Map Methods (Common for all implementation Map Objects)

1. Object put (Object key, Object value) // add entry
2. void putAll (Map m) // to add another map
3. Object get (Object key) // get value based on key
4. Object remove (Object key) // remove object based on key
5. boolean containsKey (Object key) // check for key in map
6. boolean containsValue (Object value) // check for value in map
7. boolean isEmpty () // check whether map is empty
8. int size () // return size of the map
9. void clear () // remove all entries

Views of a map

10. j. Set keySet () // convert keys of map into set
 11. k. Collection values () // convert values of map into collection
- purpose
12. l. Set entrySet () // convert whole entry into set.

Entry (I)

- * Each key-value pair is called Entry.
- * Without existence of map, there can't be existence of Entry object
- * Interface entry is defined inside Map interface.

Interface Map{

Interface Entry{

Object getKey();

Object getValue();

Object setValue(Object newValue);

}

Construction

1. Hashmap hm = new Hashmap(); // default capacity 16, loadFactor - 0.75
2. Hashmap hm = new Hashmap(int capacity);
3. Hashmap hm = new Hashmap(int capacity, float fillRatio);
4. Hashmap hm = new Hashmap(Map m);

HashTable

- * The underlying data structure for HashTable is HashTable only.
- * Duplicate keys not allowed but duplicate values allowed.
- * Order of insertion is not preserved and is based on hashCode of keys.
- * Heterogeneous objects are allowed for both keys and values.
- * 'null' insertion is not possible for both keys and values, otherwise result in NullPointerException.
- * It implements Serializable and Cloneable, but not RandomAccess.
- * Every method inside it is synchronized and thread safe.

Construction of HashTable:

1. HashTable h = new HashTable(); // initial capacity - 11, loadFactor - 0.75
2. HashTable h = new HashTable(int initialCapacity);
3. HashTable h = new HashTable(int initialCapacity, float fillRatio);
4. HashTable h = new HashTable(Map m)

- * equals(); → Hashmap } to compare content.
- * == → IdentityHashMap }

Interface new features:

- * From Java 8 we can have a method in an interface which has a body (implementation).
- * That method is by nature public and abstract, if there is a body for a method -then "default" keyword is mandatory.
- * We can override that method in implemented classes.
- * Normal interface methods need to be overridden but default methods no compulsion of overriding.
- * From Java 8 we can have a static method with a body in an interface, it will not be inherited in implementing classes.
- * To invoke static method of an interface we can use interface name.
- * From Java 9 we can write private methods with body in an interface, they will not be inherited and cannot be called by class name but they can be used within the interface.

eg: interface Hyder

```
{
    void teacher();           // public and abstract.
    void writerCode();
    default void disp() {
        System.out("Normal method allowed");
    }
}
```

Static void disp2()

```
{
    System.out("Interface Special method");
}
```

Private void disp3()

```
{
    System.out("Interface Private method");
}
```

Private void disp4()

```
{
    System.out("Interface Private method");
}
```

Class Student implements Hyder

```
{ public void teacher()
{ System.out.println("Hyder teacher java"); }
```

```
} public void writeCode()
{ System.out.println("Hyder write code"); }
```

Public class LaunchSpecialJava

```
{ public static void main (String [] args)
{ Student s = new Student();
```

```
    s.teacher();
```

```
    s.writeCode();
```

```
    s.disp();
```

```
    hyder.disp2(); }
```

```
}
```

Introduction to Generics:

Generics were introduced in Java 1.5 to provide Type-Safety and to resolve Type-Casting problems.

Type-Safety:

Arrays are always type safe that is we can give the guarantee for the type of elements present inside the array. For ex if our programming requirement is to hold String type of object, it is recommended to use String array.

In case of String array we can add only String type of object, if we try to add any other type of object, it would result in Compile-time error.

eg : String names[] = new String[500];

name[0] = "Nauin Reddy";

name[1] = "Haider";

name[2] = new Integer(10); //CE

That is we can always provide guarantee for the type of elements present inside array and hence arrays are safe to use with respect to type i.e. arrays are type safe.

But collections are not type-safe i.e. we can't provide any guarantee for the type of elements present inside collection.

For example if our programming requirement is to hold only String type of objects, it is never recommended to go for ArrayList.

If we are trying to add any other type we won't get any compile-time error but program may fail at runtime.

eg:: ArrayList al = new ArrayList();

al.add("Nauin Reddy");

al.add("Haider");

al.add(new Integer(10));

String name = (String) al.get(2); // Exception in thread "main"

Hence we can't provide guarantee for the type of elements present inside collections i.e. collections are not safe to use with respect to type.

Type-Casting (Issue)

In case of array at the time of retrieval it is not required to perform any type casting.

eg:: String name = new String[500];

name[0] = "Nauin";

String name = name[0]; // type casting not required.

But in the case of collection at the time of retrieval considering we should perform type casting otherwise we get compile-time error.

eg::

```
ArrayList al = new ArrayList();
```

```
al.add("Nawin Reddy");
```

```
String name = al.get(0); // CE
```

```
String name = (String) al.get(0); // typecasting is necessary
```

To overcome these problems of collections, Generics were introduced in 1.5v. The main objectives of generic are:

1. To provide type-safety to the collections
2. To resolve type-casting problems

To hold only String type of objects we can create a generic version of ArrayList as follows:

→ `ArrayList <String> al = new ArrayList <String>();`

```
al.add("Nawin Reddy");
```

```
al.add(10); // CE
```

If we add int type to above example results in "CE" i.e through generic we are getting type-safety. At retrieval it is not required to perform any type casting.

→ `String name = al.get(0); // type casting not required`

Conclusion:

1. Polymorphism concept is applicable only for the basic type but not for parameter type

eg:: `ArrayList <String> al = new ArrayList <String>();`

```
List <String> al = new ArrayList <String>();
```

```
Collection <String> al = new ArrayList <String>();
```

```
Collection <Object> al = new ArrayList <String>(); // CE
```

2. Collections concept only applicable for objects, parameters can be class or interface but not primitive. e.g. `ArrayList <int>` // CE

Enum:

An enum-type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.

- * Because they are constants, the names of an enum-type's field are in uppercase letters.
 - * We can write an enum either the class or outside of the class.
eg.: enum Result
- ```
{ PASS, FAIL, NR; } (internally a class is created for enum.)
```
- \* All the predefined constants (uppercase words) are static and final by default.
  - \* An enum can have methods and constructors within it.  
It can have fields → instance variables → properties

eg.: enum Result

```
{ PASS, FAIL, NR; // static final
 // PASS → public static final Result Pass = new Result();
 // FAIL → public static final Result FAIL = new Result();
 // NR → public static final Result NR = new Result();
```

Result()

```
{ System.out.println("Constructor is called"); }
```

public class LaunchEnum

```
{ public static void main(String[] args)
 { Result res = Result.PASS;
 System.out.println(res);
```

Result resArr[] = Result.FAIL.values();

```
for (Result hider: resArr)
{
 System.out.println(hider.ordinal() + ":" + hider.name());
}
```

Output: Constructor is called

PASS

0 : PASS

1 : FAIL

2 : NR.

eg:: enum Course

```
{ JAVA, JEE, SPRINGBOOT;
int courseId;
Course()
{
 System.out("Constructor");
}
```

void SetCourseId (int courseId)

```
{
 this.courseId = courseId;
}
```

int getCourseId () { return courseId; }

}

public class Launch

```
{
 public static void main (String [] args)
 {
 Course.JAVA. SetCourseID (10);
 int cid = Course.JAVA. getCourseId ();
 System.out.println (cid);
 }
}
```

## Annotations

Annotations are used to provide supplemental information about a program.

- \* Annotations start with '@'
- \* Annotations do not change the action of a compiled program.
- \* Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructor, methods, classes etc.
- \* Annotations are not pure comments as they can change the way a program is treated by the compiler.

eg:: @ FunctionalInterface

```
interface Trial { // Compiler will go to know this is
 int getNum(); } a functional interface, any changes
} would result in CE!
```

eg: class Java

```
{ public void disp() {
 System.out("Parent Display"); }
}
```

Class Focus extends Java

```
{ @Override
 public void disp() // Any changes in method signature
 { System.out("Focus is key"); } would result in Compile time
} error.
```

\* Annotation → Annotation → parent of all annotations

\* Annotations can be custom (our own)

\* Annotations can be used for class, interface, method,

instance variable, local variable, constructor, parameters, enum.

eg.: Custom annotation:

@ Target (Element.Type.TYPE)

@ Retention (RetentionPolicy.RUNTIME)

@ interface CricketPlayer

{ // @ → it's not an interface, but its annotation being created.

```
String country(); // String country () default "India"; To give
int runs(); // int runs () default 20000; initial value
}
```

// Target : to specify for what the annotation can be used, ex.

Element.Type.CLASS - can be used for class only (TYPE-for all)

// Retention: until what the annotation should be active if  
RetentionPolicy.CLASS - it will go till Jvm.

@ CricketPlayer (country = "India", runs = 20000) // to assign values

Class Viratkohli // adding annotations to annotations.

{ ... } // multiple ElementType can be added in Target.

Note: Every inbuilt annotation has a target like @Override  
can be applied to overridden methods only and cannot  
be used for class or interface.

eg.: { Viratkohli VK = new Viratkohli (); // creating Viratkohli object  
Class C = VK.getClass (); // getting the class  
Annotation an = C.getAnnotation (CricketPlayer.class); // get annot.  
CricketPlayer Cp = (CricketPlayer) an; // typecasting  
int runs = Cp.runs (); // getting the 'runs' from annotation

```

System.out.println(nns); // 20000
String cn = cp.country(); // getting country from annotation
System.out.println(cn); // India
}

```

Note: At a time single annotation can be applied to multiple places by making change at "Target".

→ @ Target ( { ElementType. METHOD, ElementType. LOCAL\_VARIABLE } )

### Generic Classes:

\* Until 1.4v a non generic version of ArrayList class is declared as follows.

#### Class ArrayList

```

add (Object o);
Object get (int index); }

```

add() method can take Object as the argument and hence we can add any type of object to the ArrayList. Due to this we are not getting type-safety. The return type of get() method is 'Object' hence at time of retrieval compulsorily we should perform type casting.

\* But in 1.5v a generic version of ArrayList class is declared as:

↓ Type parameter.

```

Class ArrayList <T>
{
 add (T t);
 T get (int index); }

```

Based on our requirement T will be replaced with our provided type. For example to hold only String type of objects we

can Create ArrayList Object as:

`ArrayList<String> l = new ArrayList<String>();`

Internally:

```
Class ArrayList<String>
{
 add (String);
 String get (int index); }
```

Add() method can take only String type as argument, hence we can add only String type objects to it, if we try to add other it would result in Compile Time Error.

eg:: `ArrayList<String> al = new ArrayList<String>();
al.add ("Naveen Reddy");
String name = al.get(0); // typecasting not required.`

\* In Generic we are associating a type parameter to the class, such type of Parameterized classes are nothing but Generic Classes.

Generic Class: class with type-parameter.

\* Based on our requirement we can create our own generic classes also.

eg:: `Class Account <T> { }
Account <Gold> g1 = new Account <Gold>();`

eg:: `Class Gen<T>
{
 T obj;
 Gen (T obj) { this.obj = obj; }`

`public void Show()
{
 System.out.println ("Type of object is " + obj.getClass().getName()); }`

```
public T getObject() { return obj; }
}
```

```
class GenericDemo {
 public static void main (String [] args) {
 Gen<Integer> g1 = new Gen<Integer>(10);
 g1.show();
 System.out.println(g1.getObject());
 }
}
```

```
Gen<String> g2 = new Gen<String>("iNeuron");
g2.show();
System.out.println(g2.getObject()); }
```

Output: The type of object is java.lang.Integer  
10

The type of object is java.lang.String  
iNeuron.

Note: To get the underlying Object of any reference type we use a method called `ref.getClass().getName()`

eg::

```
interface Calculator { }
```

```
class Casio implements Calculator { }
```

```
class Quartz implements Calculator { }
```

```
Calculator c1 = new Casio();
```

```
System.out.println(c1.getClass().getName()); // Casio
```

```
Calculator c2 = new Quartz();
```

```
System.out.println(c2.getClass().getName()); // Quartz
```

## Bounded types

We can bound the type parameter for a particular range by using extends keyword such types are called bounded types.

eg:: class Test <T> { }

Test <Integer> t1 = new Test <Integer>();

Test <String> t2 = new Test <String>();

Here as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

eg:: class Test <T extends X> { }

\* If X is a class then as the type parameter we can pass either X or its child classes.

\* If X is an interface then as the type parameter we can pass either X or its implementation classes.

eg:: class Test <T extends Number> { }

Class Demo{

    public static void main (String [ ] args)

    { Test <Integer> t1 = new Test <Integer>();

        Test <String> t2 = new Test <String>(); //CE

}

For interface also we use extends keyword



eg:: class Test <T extends Runnable> { }

Class Demo{

    public static void main (String [ ] args)

    { Test <Thread> t1 = new Test <Thread>();

        Test <String> t2 = new Test <String>(); //CE

}

Keypoints about bounded types

- \* We can't define bounded types by using implements and Super keyword.
- \* But implements keyword's purpose we can replace with extends keyword.  
 eg:: class Test < T implements Runnable > {} // invalid  
 class Test < T super String > {} // invalid.
- \* As the type parameter we can use any valid java identifier but it convention to use T always.  
 eg: class Test < T > {}  
 class Test < LineItem > {}
- \* We can pass any no. of type parameters, need not to be one.  
 eg: class Hashmap < K, V > {}  
 HashMap < Integer, String > h = new Hashmap < Integer, String >();

Which of the following are valid?

1. class Test < T extends Number & Runnable > {} // valid  
 Number - class, Runnable - interface.
2. class Test < T extends Number & Runnable & Comparable > {} // valid  
 Number - class, Runnable, Comparable - interface.
3. class Test < T extends Number & String > {} // invalid  
 We can't extend more than one class at a time.
4. class Test < T extends Runnable & Comparable > {} // valid  
 Both are interfaces so valid.
5. class Test < T extends Runnable & Number > {} // invalid  
 First class should be there then interface so invalid.

Q. Can we apply Type Parameter at Method Level?  
 Ans Yes, it is possible.

## Generic methods and wild-card character (?)

### 1. methodOne (ArrayList<String> al):

This method is applicable for ArrayList of only String type.

```
methodOne (ArrayList<String> al)
{
 al.add ("Sachin");
 al.add (new Integer(10)); // invalid
}
```

Within this method we can add only String type of objects and null to the list.

### 2. methodOne (ArrayList<?>l):

We can use this method for ArrayList of any type but within the method we can add anything to the list except null.

eg:: l.add (null); // valid  
 l.add (10); // invalid  
 l.add ("string"); // invalid

This method is useful whenever we are performing read operations.

### 3. methodOne (ArrayList<? extends X> al)

- \* X → Class, we can make a call to method by passing ArrayList of X type or its Child type.
- \* X → Interface, we can make a call to method by passing ArrayList of X type or its Implementation class.
- \* You can't add anything except null and hence best suited for read operations.

### 4. methodOne (ArrayList<? Super X> al)

- \* X → Class, we can make a call to method by passing ArrayList of X type or its Super Class.
- \* X → Interface, we can make a call to method by passing ArrayList

of  $X$  type or its Super Class of implementation class of  $X$ ,  
 method One (ArrayList<? Super X> al)  
 {  
 al.add(x); // because of only  $X$  type, you can add  
 al.add(null); }

Which of the following declarations are allowed?

1. ArrayList<String> l1 = new ArrayList<String>(); // valid
2. ArrayList<?> l2 = new ArrayList<String>(); // valid
3. ArrayList<?> l3 = new ArrayList<Integer>(); // valid
4. ArrayList<? extends Number> l4 = new ArrayList<Integer>();  
// valid
5. ArrayList<? extends Number> l5 = new ArrayList<String>();  
// invalid
6. ArrayList<?> l6 = new ArrayList<? extends Number>(); // invalid
7. ArrayList<?> l7 = new ArrayList<?>(); // invalid

Type-parameter at method level.

Class Demo<T>{

| Type parameter defined just by return type.  
 public <T> void m1(T t) { ... }  
 }

Which of the following declarations are allowed?

1. public <T> void m1(T t) // valid
2. public <T extends Number> void m1(T t) // valid
3. public <T extends Number & Comparable> void m3(T t) // valid
4. public <T extends Number & Comparable & Runnable>  
void m4(T t) // valid
5. public <T extends Number & Thread> void m5(T t) // invalid

6. Public <T extends Runnable & Number> void m6(T t) // invalid
7. Public <T extends Number & Runnable> void m7(T t) // valid

Communication with non generic code:

To provide compatibility with older versions some people compromised the concept of generic in few areas.

eg:

Class Test {

```
public static void main(String[] args)
{
 ArrayList<String> l = new ArrayList<String>();
 l.add("Sachin");
 l.add(10); // CE
```

m1(l);

l.add(10.5); // CE

System.out.println(l); // Sachin, 10, Dhoni, true

```
public static void m1(ArrayList<String> l)
{
 l.add(10);
 l.add("Dhoni");
}
```

Conclusion: Generic concept is applicable only at compile time, at runtime there is no such concept.

At the time of compilation, at the last step generic concept is removed, hence for JVM generic syntax won't be available.

Hence the following declarations are equal:

ArrayList l = new ArrayList<String>();

ArrayList l = new ArrayList<Integer>();

ArrayList l = new ArrayList<Double>();

All are equal at runtime, because compiler will remove these generic syntax. → ArrayList l = new ArrayList();

eg: `ArrayList l = new ArrayList<String>();`

`l.add(10);`

↓ RHS part is ignored.

`l.add(true);`

Since LHS is generic, so no error.

`System.out(l); // 10, true`

eg:: Class Test{

`public void m1(ArrayList<String>l){}`

`public void m1(ArrayList<Integer>l){}`

}

// CE: duplicate method found.

Behind the Scenes by the Compiler:

1. Compiler will Scan the code
2. Check the Argument type
3. If generic found in argument type then remove generic Syntax.
4. Compiler will check again the Syntax.

eg:: The Following two declarations are equal

`ArrayList<String> l1 = new ArrayList();`

`ArrayList<String> l2 = new ArrayList<String>();`

For these objects we can add only String type of objects:

`l1.add("A"); // valid`

`l1.add(10); // invalid`

### Comparable vs Comparator

`public TreeSet();`

When we use the above constructor, JVM will internally use Comparable interface method to sort the objects based on default natural sorting order.

Q1 What is Comparable interface?

Ane It is a functional interface present in java.lang package.

This interface is internally used by TreeSet Object during sorting process of the object.

@FunctionalInterface

```
public interface java.lang.Comparable<T> {
 public abstract int compareTo(T);
```

eg:: TreeSet ts = new TreeSet();

ts.add("A");

ts.add("Z");

ts.add("B");

ts.add(null); // Null pointer exception

ts.add(10); // Class Cast exception

System.out.println(ts); // [A, B, L, Z]

// Sorting of object will happen based on default natural sorting Order.

Note: If we are keeping the data inside TreeSet object, then data

should be:

a. Homogeneous → because it uses Comparable to sort the object.

b. The object should compulsorily implements an interface called "Comparable", if we fail to do so it would result in "Class Cast Except."

eg:: TreeSet ts = new TreeSet();

ts.add(new StringBuffer("A"));

ts.add(new StringBuffer("Z"));

ts.add(new StringBuffer("Y"));

System.out.println(ts); // Class Cast exception.

Note: All wrapper classes and String class have implemented "Comparable" interface. StringBuffer class has not implemented Comparable interface, so above program would result in "ClassCastException".

- \* If we are depending on default natural sorting order compulsorily Object should be Homogeneous and Comparable, otherwise results in "RE".
- \* An object is said to be Comparable if and only if corresponding class implements Comparable interface.
- \* All wrapper classes, String class already implements Comparable interface. But StringBuffer class doesn't implement Comparable interface.

### @ Functional Interface

```
public interface Comparable<T> {
 public abstract int compareTo(T); }
```

→ Obj1.compareTo(Obj2)

- returns -ve value, if obj1 has to come before obj2
- return +ve value, if obj2 has to come before obj1.
- return 0 if both are equal.

eg:: Class Test{

```
public static void main(String[] args)
{ TreeSet ts = new TreeSet();
```

ts.add("A");

ts.add("Z");

ts.add("L");

ts.add("B");

Sysout(ts); }

Rule in binary tree

-ve means node should be at left

+ve means node should be at right

Zero means nodes are duplicated

// A, B, L, Z

## Comparable (I)

Comparable interface present in java.lang package and it contains only one method CompareTo().

→ Obj1.compareTo(Obj2)

return -ve if Obj1 has to come before Obj2.

return +ve if Obj1 has to come after Obj2.

return 0 if and only if both are equal.

eg:: System.out("A".compareTo("Z")); // -ve value

System.out("Z".compareTo("K")); // +ve value

System.out("K".compareTo("K")); // zero

System.out("R".compareTo(null)); // NullPointerException

- \* Whenever we are depending on default natural sorting order and if we are trying to insert elements then internally JVM will call CompareTo() to identify sorting order.
- \* For 'String' and 'Number' natural sorting order is ascending order.

## Comparator (I)

If we are not satisfied with default sorting order or if default natural sorting order is not already available then we can define our own sorting by using Comparator Object.

→ Public interface java.util.Comparator<T> {

    Public abstract int compare(T, T);

    Public abstract boolean equals(java.lang.Object); }

- \* Comparator (I) is present in java.util package. It contains two methods - compare() and equals().

→ public int compare (Object obj1, Object obj2)

returns -ve if obj1 comes before obj2

returns +ve if obj1 comes after obj2.

returns 0 if obj1 and obj2 are equal

→ public boolean equals (Object o);

Whenever we are implementing Comparator Interface Compulsory  
we should provide implementation for compare().

Implementing equals() is optional because it is already  
available to our class from Object class through Inheritance.

eg:: Class TreeSetDemo {

    public static void main (String [] args)

    { TreeSet t = new TreeSet (new MyComparator()); // Line 1

        t.add (10);

        t.add (0);

        t.add (5);

        t.add (15);

        t.add (20); // 20, 15, 10, 5, 0

    System.out.println (t); }

}

Class MyComparator implements Comparator {

    public int compare (Object obj1, Object obj2)

    { Integer i1 = (Integer) obj1;

        Integer i2 = (Integer) obj2;

        if (i1 < i2)

            return +1;

        else if (i1 > i2) return -1;

        else return 0; }

}

- \* If we are not passing Comparator object as an argument then internally JVM will call compareTo(), in that case output will be [0, 5, 10, 15, 20].
- \* At line 1 if we are passing Comparator object then JVM will call compare() instead of compareTo() which is meant for customized sorting.

```
TreeSet ts = new TreeSet(new MyComparator());
```

```
ts.add(10);
```

```
ts.add(0); Compare (0, 10)
```

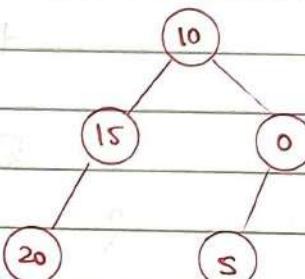
```
ts.add(15); Compare (15, 10)
```

```
ts.add(5); Compare (5, 10)
```

Compare (5, 0)

ts.add(20); Compare (20, 10)

Compare (20, 0)



→ 20 15 10 5 0

### Various Possible implementations of compare()

```
public int compare(Object obj1, Object obj2)
{
 Integer i1 = (Integer) obj1;
 Integer i2 = (Integer) obj2;
 return i1.compareTo(i2); // ascending
 return -i1.compareTo(i2); // descending
 return i2.compareTo(i1); // descending
 return -i2.compareTo(i1); // ascending
 return +1; // insertion order
 return -1; // reverse of order of insertion
} return 0; // only first element.
```

- \* W.A.P. - To insert String Objects into the TreeSet where the Sorting Order is of reverse of alphabetical order.

→

```

import java.util.*;
class TreeSetDemo {
 public static void main(String[] args) {
 TreeSet t = new TreeSet(new MyComparator());
 t.add("Sachin");
 t.add("Pointing");
 t.add("Sanakara");
 t.add("Firmin");
 t.add("Iara");
 System.out.println(t);
 }
}

```

```

class MyComparator implements Comparator {
 public int compare(Object obj1, Object obj2) {
 String s1 = (String) obj1;
 String s2 = obj2.toString();
 return s2.compareTo(s1);
 }
}

```

- \* W.A.P to insert StringBuffer Objects into the TreeSet where Sorting order is Alphabetical Order.

→

```

import java.util.*;
class TreeSetDemo {
 public static void main(String[] args) {
 TreeSet t = new TreeSet(new MyComparator1());
 }
}

```

```

 t.add (new StringBuffer ("A"));
 t.add (new StringBuffer ("Z"));
 t.add (new StringBuffer ("K"));
 System.out.println(t);
}
}

```

```

Class MyComparator implements Comparator {
 public int compare (Object obj1, Object obj2)
 {
 String s1 = obj1.toString();
 String s2 = obj2.toString();
 return s1.compareTo(s2); } // A, K, Z
 }
}

```

- \* W.A.P to insert String and StringBuffer Objects into the TreeSet where sorting order is increasing length order. If two Objects have same length then consider their alphabetical order



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

```
class TreeSetDemo {
```

```

 public static void main (String [] args)
 {
 TreeSet t = new TreeSet (new MyComp ());
 t.add ("A");
 t.add (new StringBuffer ("ABC"));
 t.add (new StringBuffer ("AA"));
 t.add ("xx");
 t.add ("A");
 System.out.println(t);
 }
}

```

```

Class MyComp implements Comparator
{

```

```

 public int compare (Object obj1, Object obj2) {
```

```

String s1 = Obj1.toString();
String s2 = Obj2.toString();
if int i1 = s1.length();
int i2 = s2.length();
if (i1 < i2) return -1;
else if (i1 > i2) return 1;
else return s1.compareTo(s2);
}

```

Note: \* If we use TreeSet, then Condition is :

- Object should be homogenous
- Object should be Comparable (Class should implement Comparable (I)).

\* If we use TreeSet (Comparator C) then :

- Object need not be homogenous
- Object need not implement Comparable (I).

Q.1. When to go for Comparable (I) and when to go for Comparator (I)?

Ans Predefined Comparable Classes like String, wrapper class for which default natural sorting is available, if we are not satisfied with natural sorting order or we need to modify then we need to go for Comparator (I).

\* For predefined Non-Comparable Class like StringBuffer, Comparator is used for both natural sorting order and customized sorting order.

\* For User-defined Class like Employee, Student, the developer if he comes up with own logic of sorting, then he should implement Comparable (I) and give it as a ready made logic.

Class Employee implements Comparable

```
{ int id;
 String name;
 int age;
```

```
public int compareTo(Object obj) {
```

// Sorting is done based on "id"

```
 ;
```

```
}
```

If the developer is using Employee class, if he is not interested in sorting based on "id" given by the API then he can use "Comparator".

### Comparable v/s Comparator (Usage)

- For predefined Comparable classes (like String) default natural sorting order is already available. We can customize it by defining our own sorting by Comparator object.
  - For pre-defined non-comparable classes (like StringBuffer) default natural sorting order is not available, if we want to define our own sorting we can use Comparator object.
  - For our own classes (like Employee) the person writing Employee class is responsible to define sorting order by implementing Comparable interface (default).
  - The person who is using our (Employee) class wants to modify the order of sorting then he can define his own sorting order by using Comparator object.
- \* W.A.P to insert Employee Objects into the tree set where DSC is based on ascending Order of Employee Id and Customized sorting order is based on alphabetical Order of name.

→ import java.util.\*;

```

class Employee implements Comparable {
 String name;
 int eid;
}

Employee (String name, int eid)
{
 this.name = name;
 this.eid = eid;
}

public String toString() { return name + " " + eid; }

public int compareTo (Object obj)
{
 int eid1 = this.eid;
 Employee e = (Employee) obj;
 int eid2 = e.eid;
 if (eid1 < eid2) return -1;
 else if (eid1 > eid2) return 1;
 else return 0;
}

```

Class Test{

```

public static void main (String [] args)
{
 Employee e1 = new Employee ("sachin", 10);
 Employee e2 = new Employee ("pointing", 14);
 Employee e3 = new Employee ("lara", 9);
 Employee e4 = new Employee ("anwar", 23);
 Employee e5 = new Employee ("faintoff", 17);
}

```

TreeSet t = new TreeSet();

```

t.add (e1); t.add (e2);
t.add (e3); t.add (e4);
t.add (e5);

```

System.out.println (t);

Output:

Lara 9

Sachin 10

pointing 14

faintoff 17

Anwar 23

```

TreeSet t1 = new TreeSet(new MyComparator());
t1.add(e1); t1.add(e2);
t1.add(e3); t1.add(e4);
t1.add(e5);
System.out.println(t1);
}

```

answar 23

fintoff 17

lara 9

pointing 14

Sachin 10

Output

Class MyComparator implements Comparator

```

public int compare(Object obj1, Object obj2)
{
 Employee e1 = (Employee) obj1;
 Employee e2 = (Employee) obj2;
 String s1 = e1.name;
 String s2 = e2.name;
 return s1.compareTo(s2);
}

```

### Comparable (I)

- Present in java.lang package
- It is meant for default natural sorting order
- Defines only one method: compareTo()
- All wrapper class and String class implements Comparable interface

### Comparator (I)

- Present in java.util package
- It is meant for customized sorting order.
- Defines two methods: compare() and equals()
- The only implemented classes of Comparator are Collator and RueBaseCollator.

## Functional Interface

If an interface contains only one abstract method then such interfaces are called as "Functional Interface".

```
public interface java.util.function.Predicate<T>
{
 public abstract boolean test(T);
 // default methods
}
```

public java.util.function.Predicate<T> and (java.util.function.Predicate<? Super T>);

public java.util.function.Predicate<T> negate();

public java.util.function.Predicate<T> or (java.util.function.Predicate<? Super T>);

// Static method

public static <T> java.util.function.Predicate<T> isEqual
 (java.lang.Object);

}

## Use of Predicate:

Class MyPredicate implements Predicate<Integer>

```
{
 @Override
 public boolean test(Integer i)
 {
 if (i > 10)
 return true;
 else
 return false;
 }
}
```

\* Instead of writing a separate class we can write Lambda Expression:

→ Predicate<Integer> p = i → i > 10;

Sysout (p.test(10)); // false

Sysout (p.test(100)); // true.

\* Write a predicate to check whether the given String length is  $\geq 3$

```
Class MyPredicate implements Predicate<String>
{
 @Override
 public boolean test(String name)
 {
 if (name.length() ≥ 3)
 return true;
 else
 return false;
 }
}
```

Lambda Expression:

```
Predicate<String> p = name \rightarrow name.length() ≥ 3 ;
System.out.println(p.test("PNC")); // true
System.out.println(p.test("CS")); // false
```

Default methods available as utility methods:

```
public default Predicate<T> and (Predicate p);
public default Predicate<T> negate ();
public default Predicate<T> or (Predicate p);
```

```
import java.util.function.*;
public class Test
{
 public static void main (String [] args)
 {
 int [] arr = { 0, 5, 10, 15, 20, 25, 30 };
 Predicate<Integer> p1 = i \rightarrow i ≥ 10 ;
 System.out.println ("Elements greater than 10 are : ");
 m1 (p1, arr); // 15, 20, 25, 30
 }
}
```

```
Predicate<Integer> p2 = i \rightarrow i % 2 == 0;
System.out.println ("Even numbers are : ");
m1 (p2, arr); // 0 10 20 30
```

```
Sysout("Elements greater than 10 and even are :");
m1(p1.and(p2), arr); // 20 30
```

```
Sysout("Elements greater than 10 or even are :");
m1(p1.or(p2), arr); // 0 10 15 20 25 30
```

```
Sysout("Elements not even are :");
m1(p2.negate(), arr); } // 5 15 25
```

```
public static void m1(Predicate<Integer> p, int[] x)
{
 for (int ele : x)
 if (p.test(ele))
 Sysout(ele);
}
```

### Function (I)

```
public interface java.util.function.Function<T, R> {
 // 1 abstract method
 public abstract R apply(T);
 // default method
 public <V> java.util.function.Function<V, R>
 compose(java.util.function.Function<? Super V, ? extends T>);

 public <V> java.util.function.Function<T, V> andThen
 (java.util.function.Function<? Super R, ? extends V>);

 // static method
 public static <T> java.util.function.Function<T, T> identity();
}
```

Writing a code using Implementation class

```
Class MyFunction implements Function<String, Integer>
{
 @Override
 public Integer apply (String name)
 {
 return name.length();
 }
}
```

Public Class Test{

```
public static void main (String [] args)
{
 Function<String, Integer> f = new MyFunction ();
 int output = f.apply ("Sachin");
 System.out.println (output);
 System.out.println ("Sachin".length ());
}
```

Above Code with Lambda Expression:

public class Test{

```
public static void main (String args[])
{
 Function<String, Integer> f = name -> name.length();
 int output = f.apply ("Sachin");
 System.out.println (output);
}
```

Note: When to go for Predicate and when for Function?

- \* **Predicate:** To implement some conditional checks we should go for Predicate.

- \* **Function:** To perform some operation and to return some result we should go for Function.

## Method Reference (::) and Constructor reference (::)

:: → Scope resolution Operator.

Syntax for method reference

1. static method

Class Name :: methodName

2. instance method

Object :: methodName

Eg:: public class Test{

public static void m1(){

for(int i=1; i<=10; i++)

{ System.out.println("Child thread"); }

}

public static void main(String[] args) throws Exception{}

// Using method reference binded the method call of m1() of interface 'Runnable'

Runnable r = Test :: m1;

Thread t = new Thread(r);

t.start();

for(int i=0; i<10; i++)

{ System.out.println("main thread"); }

}

}

Output:

Child thread

Child thread

main thread

main thread

:

:

eg.: interface Intef

```
{ public void m1(int i) }
```

public class Test{

```
public void logic(int i) { System.out("method ref."); }
```

```
public static void main(String []args)
{ Intef i = a → System.out("lambda expression");
 i.m1(100); }
```

```
Intef ii = new Test() :: logic;
ii.m1(10); }
```

Output:

"lambda expression"  
"method ref."

eg.: Constructor reference

Class Sample

```
{ private String s;
 Sample (String s) { this.s=s; System.out("Constructor executed");
 System.out("Constructor executed"); }
 }
```

@FunctionalInterface

interface Intef

```
{ public Sample get(String s); }
```

public class Test{

```
public static void main(String []a)
```

```
{ Intef i = s → new Sample (s); }
```

```
i.get("from lambda expression..."); }
```

## // Constructor reference

```
Interf ii = Sample :: new;
ii.get("from constructor reference"); }
```

{

### Usage of "forEach()" to print elements of ArrayList.

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

```
import java.util.function.*;
```

```
// public void forEach(java.util.function.Consumer<? super E>);
```

```
// public abstract void accept(T t)
```

```
Class MyConsumer implements Consumer<String>
```

```
{ public void accept(String name){
```

```
System.out("accept method call");
```

```
System.out(name); }
```

{

```
public class Test{
```

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

```
ArrayList<String> names = new ArrayList<String>();
```

```
names.add("Sachin"); names.add("Chori");
```

### // Traditional approach

```
Consumer<String> consumer = new MyConsumer();
```

```
names.forEach(consumer);
```

### // Lambda expression

```
names.forEach(name → System.out.println(name));
```

### // method reference

```
names.forEach(System.out::println); }
```

{

## Stream API

- \* Stream: Channel through which there is a free flow movement of data.
- \* Streams: To process Objects of the Collection, in 1.8v Streams Concept was introduced.

Q1. Difference between Java.util.Streams and java.io.Streams.

Ans Java.util Streams meant for processing objects from the Collection. That is it represents a Stream of objects from the collection. But java.io Streams meant for processing binary and character data with respect to file that is it represents stream of binary data or character data from the file, hence both Java.io Streams and Java.util Streams both are different.

Q2. Difference between Collection and Stream?

- \* If we want to represent a group of individual Objects as a single entity then we should go for collection.
- \* If we want to process a group of objects from the collection then we should go for Stream.
- \* We can Create a Stream Object to the Collection by Using Stream() method of Collection Interface. Stream() method is a default method added to Collection in 1.8v

```
import java.util.*;
import java.util.stream.*;
public class Test{
 public static void main (String [] args) {
 ArrayList < Integer > al = new ArrayList < Integer > ();
 al.add(0); al.add(5); al.add(10);
 al.add(15); al.add(20); al.add(25);
 System.out.println(al); // [0, 5, 10, 15, 20, 25]
 }
}
```

// Till JDK 1.7 v

```
ArrayList<Integer> evenlist = new ArrayList<Integer>();
for (Integer ii : al)
 if (ii % 2 == 0)
 evenlist.add(ii);
System.out.println(evenlist); // [0, 10, 20]
```

// From JDK 1.8 v we use Streams

```
// Configuration → al.stream()
// Processing → filter(i → i % 2 == 0).collect(Collectors.toList())
List<Integer> Streamlist = al.stream().filter(i → i % 2 == 0)
 .collect(Collectors.toList());
System.out.println(Streamlist); // 0 10 20
Streamlist.forEach(System.out::println); {
}
// 0
// 10
// 20
```

eg:: (for the above code, continuation)

// Till JDK 1.7 v

```
ArrayList<Integer> doublelist = new ArrayList<Integer>();
for (Integer ii : al)
 doublelist.add(ii * 2);
System.out.println(doublelist); // 0 10 20 30 40 50
```

// from JDK 1.8 v

// map - for every Object if a new Object has to be created thus  
go for map.

```
List<Integer> Streamlist = al.stream().map(Obj → Obj * 2)
 .collect(Collectors.toList());
System.out.println(Streamlist); // 0 10 20 30 40 50
Streamlist.forEach(i → System.out.println(i));
System.out.println(); { 0 10 20 30 40 50 }
```

- \* Stream is an interface present in java.util.stream. Once we get the Stream, by using that we can process objects of that collection.  
We can process the objects in 2 phases:
1. Configuration
  2. Processing.

### ① Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering: We can configure a filter to filter elements from the collection based on some boolean condition by using filter() method of Stream interface.

→ public Stream filter (Predicate<T> t)

Here (Predicate<T> t) can be a boolean valued function / lambda expression.

eg.: Stream S = C.stream();

Stream S<sub>1</sub> = S.filter (i → i+2 == 0);

Hence to filter elements of collection based on some boolean condition we should go for filter method.

Mapping: If we want to create a separate new object for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

→ public Stream map (Function f);

It can be lambda expression also.

eg.: Stream S = C.stream();

Stream S<sub>1</sub> = S.map (i → i+10);

Once we perform operations we can process objects by using several methods.

2.

## Processing:

Processing can be done by multiple methods:

1. Collect() method
2. Count() method
3. Sorted() method
4. min() and max method
5. forEach() method
6. toArray() method
7. Stream.of() method



```

eg:: import java.util.*;
import java.util.stream.*;

public class Test{
 public static void main(String []args)
 {
 ArrayList<String> names = new ArrayList<String>();
 names.add("Sachin"); names.add("Harbhajan");
 names.add("Dhoni"); names.add("Kohli");

 List<String> result = names.stream().filter(name ->
 name.length() > 5).collect(Collectors.toList());
 System.out.println(result.size()); //2
 }
}

```

```

long count = names.stream().filter(name ->
 name.length() > 5)
 .count();
System.out.println(count); //2

```



```

eg:: import java.util.*;
import java.util.stream.*;
// Comparable (Predefined API for natural sorting order)
 → compareTo(Object obj)

```

// Comparator (for user-defined class for customized sorting orders)  
→ compare (Obj1, Obj2)

## Public Class Test

public static void main (String [large])

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

```
al.add(10); al.add(0); al.add(15);
```

al.add(5); al.add(20);

```
Sysout ("Before sorting:" + al); // 10 0 15 5 20
```

## // Using Stream API

```
List<Integer> result = al.parallelStream().sorted().collect(Collectors.toList());
```

```
Sysout ("After Sorting :::" + result);
```

// After sorting 0 5 10 15 20

List<Integer> custom = al.stream().sorted((i1, i2) →

i2. CompareTo(i1).Collect(Collectors.toList());

```
Sysout ("After Sorting ::" + custom); }
```

// After Sorting 20 15 10 5 0

2

```
eg:: ArrayList < Integer > al = new ArrayList < Integer > ();
al.add(10); al.add(0); al.add(15);
al.add(5); al.add(20);
System.out.println(al); // 10 0 15 5 20
```

Object[] ObjArr = al.stream().toArray();

for (Object Obj : ObjArr)

Sysout(Obj); // 10 0 15 5 20

```
Integer [] ObjArr = a1.stream().toArray(Integer []::new);
```

for ( Integer i : ObjArr ) System.out(i);

// 10015520

(i) Collect() : This method collects the elements from the stream and adding to the specified to collection indicated by the argument.

eg:: { ArrayList<String> names = new ArrayList<String>();  
 names.add("Sachin"); names.add("Saurav");  
 names.add("Dhoni"); names.add("Yuvi"); }

List<String> result = names.stream().filter(name → name.length() > 5).collect(Collectors.toList());  
 System.out(result);

List<String> mapR = names.stream().map(name → name.toUpperCase()).collect(Collectors.toList());  
 System.out(mapR);

{}

(ii) Count() : This method returns number of elements present in the Stream.

→ public long count()

eg:: { ArrayList<String> names = new ArrayList<String>();  
 names.add("Sachin"); names.add("Kohli");  
 names.add("Yuvaraj"); }

long count = names.stream().filter(name → name.length() > 5).count();

System.out(count); }

(iii) sorted(): If we want to sort the elements present inside Stream  
then we should go for sorted() method.

sorted() → default natural sorting Order

sorted(Comparator c) → Customized sorting order

eg:: {  
 ArrayList<Integer> al = new ArrayList<Integer>();  
 al.add(10); al.add(20); al.add(0);  
 al.add(5); al.add(25); al.add(15);

List<Integer> result = al.stream().sorted().collect(Collectors.  
 toList());  
 System.out(result);

List<Integer> custom = al.stream().sorted((i1, i2) → i1.compareTo  
 (i2)).collect(Collectors.toList());  
 System.out(custom); }

(iv) min() and max():

min(Comparator c) returns min value according to specified  
comparator.

max(Comparator c) returns maximum value according to  
specified comparator.

eg:: {  
 ArrayList<Integer> al = new ArrayList<Integer>();  
 al.add(75); al.add(15); al.add(3); al.add(0);

Integer mn = al.stream().min((i1, i2) → i1.compareTo(i2)).get();  
 System.out(mn);

Integer mx = al.stream().max((i1, i2) → i1.compareTo(i2)).get();  
 System.out(mx); }

(v) forEach() : This method will not return anything. It will take lambda expression as argument and apply that lambda expression to each element present in stream.

eg:: { ArrayList<String> al = new ArrayList<String>();  
 al.add ("Sachin");  
 al.add ("Kohli"); }

al.stream().forEach (a1 → System.out.println()); // Sachin Kohli  
 al.stream().forEach (System.out::println); }  
 // Sachin  
 // Kohli

(vi) toArray() : We can make use of toArray() method to copy elements present in the Stream into specified array.

eg:: { ArrayList<Integer> al = new ArrayList<Integer>();  
 al.add (10); al.add (5); }

Integer [ ] arr = al.stream().toArray(Integer [ ] :: new);  
 for (Integer element : arr)  
 System.out.println(element); } // 10 5

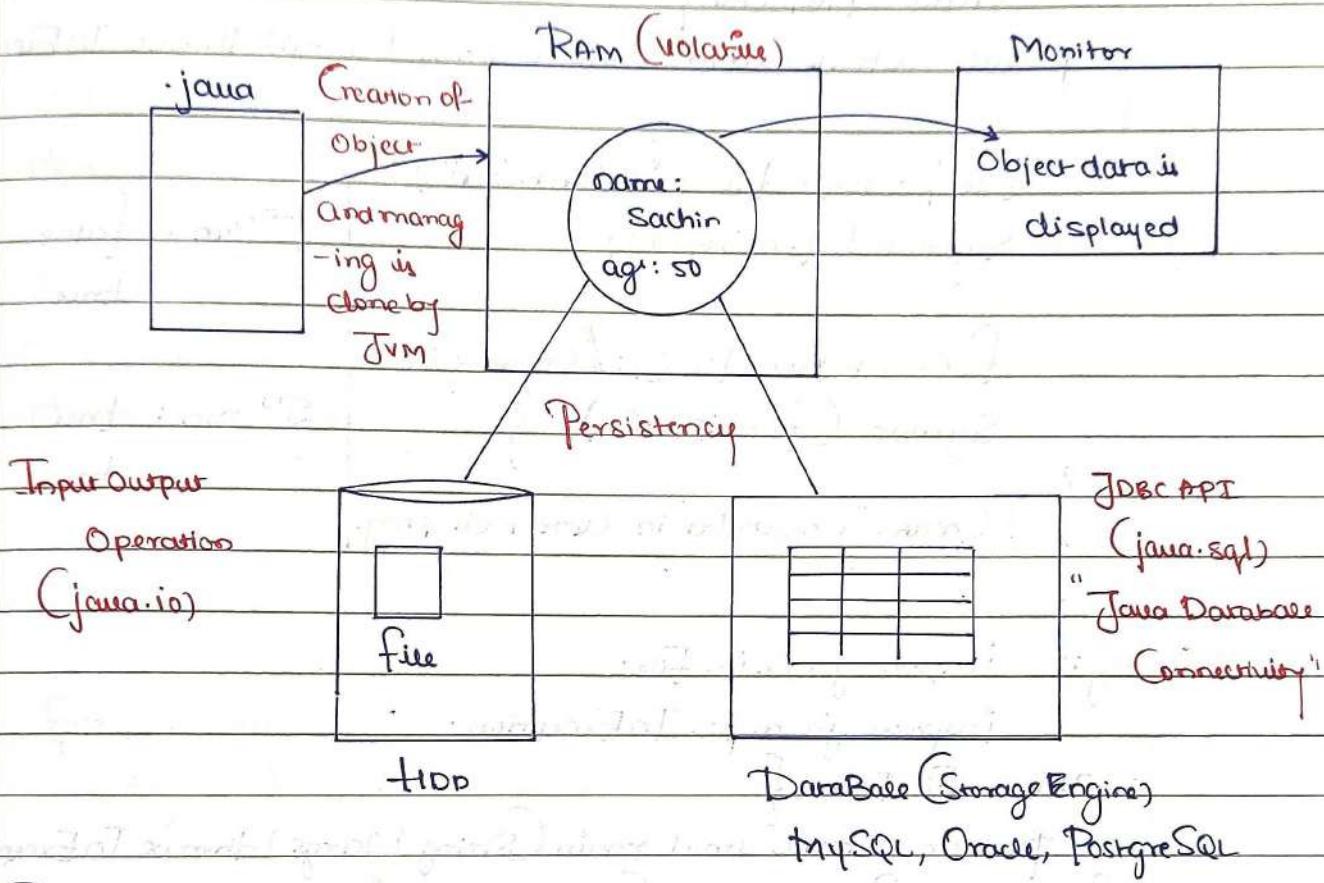
(vii) Stream.of() : We can also apply a Stream for group of values and for arrays.

eg:: { Stream s = Stream.of (99, 999, 9999);  
 s.forEach (System.out::println); }

Double [ ] d = { 10.0, 10.1, 10.2, 10.3 };

Stream s1 = Stream.of (d);  
 s1.forEach (System.out::println); }

## File Operation in Java



### Persistence:

It is a mechanism of storing the data permanently on to the file. In java persistence can be achieved through an API's available inside package called "java.io".

### File:

```
File f = new File("abc.txt");
```

- \* The line first checks whether abc.txt file is already available or not, if it is already available then "f" simply refers that file.
- \* If it is not already available then it won't create any physical file, just creates a java file object represents the name of the file.
- \* A java file object can represent a directory also.

→ eg:: import java.io.\*;  
 Class FileDemo{  
 public static void main (String []args) throws IOException  
 {  
 File f = new File ("cricket.txt");  
 System.out.println (f.exists());  
 f.createNewFile(); // Create file  
 System.out.println (f.exists()); }  
} // Create "cricket.txt" in current directory.

|                      |               |
|----------------------|---------------|
| 1 <sup>st</sup> Run: | false<br>true |
| 2 <sup>nd</sup> run: | true<br>true  |

→ eg:: import java.io.File;  
 import java.io.IOException;  
 Class FileDemo{  
 public static void main (String []args) throws IOException  
 {  
 File f = new File ("IPLTeam"); // no extension → directory  
 System.out.println (f.exists());  
 f.mkdir(); // Create directory  
 System.out.println (f.exists()); }  
} // Create a directory in  
current directory. (failure)

|                      |               |
|----------------------|---------------|
| 1 <sup>st</sup> Run: | false<br>true |
| 2 <sup>nd</sup> Run: | true<br>true  |

Note: In UNIX everything is a file, Java "file I/O" is based on UNIX Operating System hence in Java also we can represent both files and directories by file object only.

## File Class Constructors:

1. `File f = new File(String name);`

→ Create a java file object that represents name of the file or directory in current working directory.

2. `File f = new File(String SubDirName, String name);`

→ Create a file object that represents name of the file or directory present in Specified Sub directory.

eg:: `File f1 = new File("abc");`

`f1.mkdir();`

`File f2 = new File("abc", "demo.txt");`

3. `File f = new File(File Subdir, String name);`

eg:: `File f1 = new File("abc");`

`f1.mkdir();`

`File f2 = new File(f1, "demo.txt");`

→ eg:: `class Test{`

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

`{`

`File f = new File("ineuron");`

`f.mkdir();`

`System.out.println(f.isDirectory()); // true`

*// Create a "ineuron"*

`File f1 = new File(f, "abc.txt");`

`f1.createNewFile();`

`System.out.println(f1.isFile()); } // true`

*directory in current working directory and a file "abc.txt" is created inside it //*

`}`

*// If "ineuron" directory is not available and you try to create file inside it, it would result in "FileNotFoundException".*

```

eg:: { String location = "c://pwskills";
 File f = new File (location); //Creates the "pwskills"
 f.mkdir (); directory in C drive
 and the "java.txt"
 File fi = new File (f, "java.txt"); fi is created inside
 fi.createNewFile (); } it. //

```

### Important methods of file class:

1. boolean exists(): returns true if physical file or directory available.
2. boolean CreateNewfile(): this first checks if physical file is already available or not, if it is already available then it returns false without creating any physical file. If file is not already available then it creates new file and returns true.
3. boolean mkdir(): this method checks if directory already available or not, if available then it returns false and returns without creating anything. If directory not already available then it will create new directory and return true.
4. boolean isFile(): returns true if file object represents a physical file.
5. boolean isDirectory(): returns true if file object represents a directory.
6. String [] list(): returns names of all files and subdirectories present in specified directory.
7. long length(): returns number of characters present in a file.
8. boolean delete(): to delete a file or a directory.

Write a program to display the names of all files and directories present in "D://Java Job Guaranteed Batch" and return the number of files and directories present inside.

```

import java.io.*;
class Test{
 public static void main(String []args) throws Exception
 {
 int dirCount=0;
 int jpgFileCount=0;
 int txtFileCount=0;
 int zipFileCount=0;
 String location = "D:\\Java Job Guranteed Batch";
 File f = new File(location);
 String [] names = f.list();
 for (String name: names)
 {
 // iterating over all files in location folder
 File fi = new File(f, name); // creating new file for every file to
 if (fi.isDirectory()) dirCount++; // apply 'is File()' method
 // new file not created if it's already
 if (fi.isFile())
 {
 if (name.endsWith(".jpg")) jpgFileCount++;
 if (name.endsWith(".txt")) txtFileCount++;
 if (name.endsWith(".zip")) zipFileCount++;
 }
 System.out.println(name);
 }
 System.out.println(jpgFileCount);
 System.out.println(txtFileCount);
 System.out.println(zipFileCount);
 System.out.println(dirCount);
 }
}

```

// in loop new file are not created  
 as they are already available, but  
 it will point to already available  
 file and file methods can be  
 applied.

FileWriter : By using FileWriter Object we can write character data to the file.

→ FileWriter fw = new FileWriter (String name);

→ FileWriter fw = new FileWriter (File f);

The above two Constructors meant for overiding the data to the file.

\* Instead of overiding if we want append operation then we should go for :

→ FileWriter fw = new FileWriter (String name, boolean append);

→ FileWriter fw = new FileWriter (File f, boolean append);

\* If the specified physical file is not already available then these Constructors will create that file.

#### Methods:

1. write (int ch) : to write a single character to the file.

2. write (char[] ch) : to write an array of characters to the file.

3. write (String s) : to write a string to the file.

4. flush () : to give the guarantee the total data includes does characters also written to the file.

5. close () : to close the stream.

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

```
class Test{
```

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

```
{
```

```
 File fi = new File ("abc.txt");
```

```
 FileWriter fw = new FileWriter (fi); // write to fi file
```

// performing write operation on a file

```
 fw.write (97); // 'a' (ascii value of 97)
```

```

f2.write("\n"); // adds a new line | acts like 'Enter' key
f2.write("Hello World!"); // to add string
char [7ch = { 'V', 'i', 'N', 'E', 'K' };
f2.write(ch); // passing char array
f2.flush(); // making the data to write to the file.
f2.close(); } // to close the resource
}

```

A new "abc.txt" file is created in Current Working Directory.

Note: If "abc.txt" already available and if you run this code then the content will be overridden.

To append Change → new FileWriter(f, true); this will now append all the writings to the available file.

The main problem with FileWriter is we have to insert line separator manually, which is difficult to the programmer ('\n'), and line separator varying from system to system.

File Reader: By using FileReader Object we can read Character data from the file.

- \* FileReader fr = new FileReader(String name);
- \* FileReader fr = new FileReader(File f);

#### METHODS:

1. int read(): it attempts to read next character from the file and return its unicode value. If next character is not available then we will get -1.  
eg:: int i = fr.read();

2. int read(char[7ch]): it attempts to read enough characters from the file into char[7] array and return the number of characters copied from the file into Char Array.

eg:: import java.io.FileReader;  
 import java.io.IOException;

public class Test{

```
 public static void main (String [] args) throws IOException
 {
 FileReader fr = new FileReader ("abc.txt");
 int i = fr.read();
 while (i != -1)
 {
 System.out ((char) i); // typecasting as i is Unicode
 i = fr.read(); } // prints all the characters from file
 }
```

eg:: class Test{

```
 public static void main (String [] args) throws Exception
 {
 File f = new File ("sachin.txt");
 FileReader fi = new FileReader (f);
 }
```

Char [] ch = new Char [(int) f.length ()];

// f.length () is long so typecasting to int, making  
 char array of the no. of characters in the file.

fi.read (cb); // reads and stores chars inside array

for (char data: ch)

System.out (data); // accessing the characters from array

fi.close ();

} // Closing the resources

}

Usage of FileWriter and FileReader is not recommended because of following reasons:

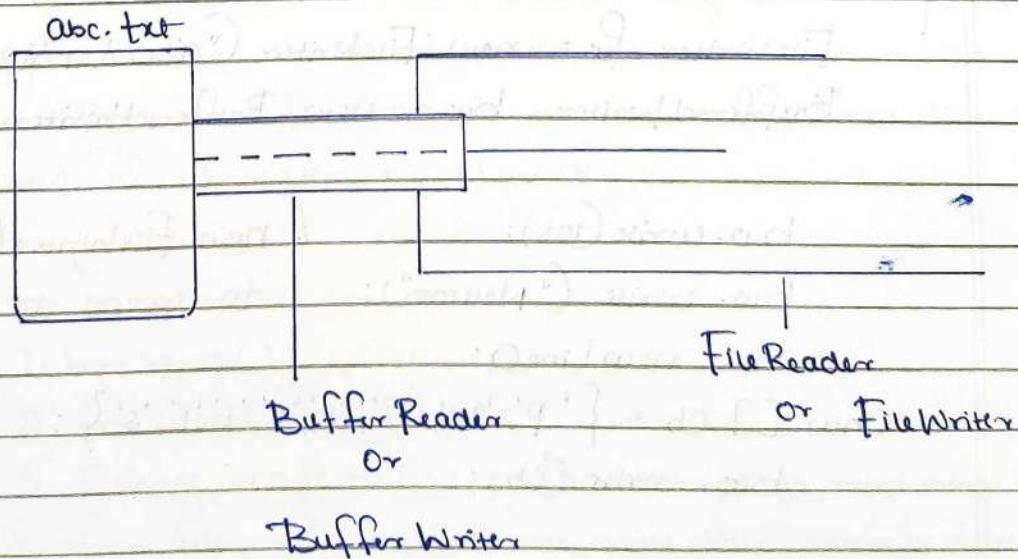
1. While writing data by FileWriter compulsory we should insert line separator ('\n') manually which is a big headache to programmer.
2. While reading data by FileReader we have to read character by character instead of line by line which is not convenient.  
eg:: we need to search for a 10 digit mobile no. present in file called 'mobile.txt', Since we can read only characters just to search one number 10 searching is required and to search 10,000 mobile numbers we need to read 1Cr times, so performance is very low.
3. To overcome these problems we should go for "BufferedReader" and "BufferedWriter."

### BufferedWriter:

By using BufferedWriter we can write the character data to the file. It can't communicate with the file directly, it can communicate only with Writer Object.

#### Constructors:

- \* BufferedWriter bw = new BufferedWriter (Writer w);
- \* BufferedWriter bw = new BufferedWriter (Writer w, int size);



Which of the following declarations are valid?

1. new BufferedWriter ("cricket.txt"); // invalid
2. new BufferedWriter (new File ("cricket.txt")); // invalid
3. new BufferedWriter (new FileWriter ("cricket.txt")); // valid
4. new BufferedWriter (new BufferedWriter (new FileWriter ("cricket.txt"))); // valid

### Methode :

1. write (int ch);
2. write (char [ ] ch);
3. write (String s);
4. flush();
5. close();
6. newline(); → for inserting a new line character to the file.

Q1 When compared to FileWriter which of the following Capability is available as a method in BufferedWriter.

Ans Inserting new line character → newline();

e.g.: import java.io.\*;

Class Test {

public static void main (String [ ] args)

{

FileWriter fw = new FileWriter ("abc.txt");

BufferedWriter bw = new BufferedWriter (fw);

bw.write (105);

// new FileWriter ("abc.txt", true);

bw.write ("Neuron");

to append to file.

bw.newLine();

char [ ] ch = { 'P', 'W', 'S', 'K', 'I', 'L', 'L', 'S' };

bw.write (ch);

```

bw.newLine();
bw.write("Unicorn");

bw.flush(); // make sure the operation is successful or fail
bw.close(); // internally fw.close() will happen.

}
}

```

Note:

1. bw.close(); // recommended to use
2. fw.close(); // not recommended
3. bw.close(); fw.close(); // not recommended

Whenever we are closing BufferedWriter automatically underlying writer will be closed and we don't close explicitly.

Buffered Reader:

This is the enhanced (better) reader to read character data from the file.

Constructors:

- \* BufferedReader br = new BufferedReader(Reader r);
- \* BufferedReader br = new BufferedReader(Reader r, int size);

BufferedReader cannot communicate directly with the file, it should communicate via some reader object. The main advantage of BufferedReader over FileReader is we can read data line by line instead of character by character.

Methode:

1. int read();
2. int read(char [ ] ch);
3. String readLine(); → It attempts to read next line and returns it from file, if next line not avail then returns null.

```

eg:: import java.io.*;
public class Test{
 public static void main (String [] args) throws IOException
 {
 FileReader fr = new FileReader ("abc.txt");
 BufferedReader br = new BufferedReader (fr);
 String line = br.readLine ();
 while (line != null)
 {
 System.out.println (line);
 line = br.readLine ();
 }
 br.close ();
 }
}

```

Note:

1. br.close(); // recommended
2. fw.close(); // not recommended
3. br.close(); fw.close(); // not recommended to use both.

Whenever we are closing BufferedReader automatically underlying FileReader will be closed, not required to close explicitly.

PrintWriter:

This is the most enhanced writer to write text data to the file. By using FileWriter and BufferedWriter we can write only character data to the file but by using PrintWriter we can write any type of data to the file.

Constructors:

PrintWriter pw = new PrintWriter (String name);

PrintWriter pw = new PrintWriter (File f);

PrintWriter pw = new PrintWriter (Writer w);

## Methods:

- |                        |                          |
|------------------------|--------------------------|
| 1. write (int ch);     | 10. print (String s);    |
| 2. write (char [7ch]); | 11. println (char ch);   |
| 3. write (String s);   | 12. println (double d);  |
| 4. flush ();           | 13. println (int i);     |
| 5. close ();           | 14. println (boolean b); |
| 6. print (char ch);    | 15. println (Strings);   |
| 7. print ("Int");      | 16. println (String s);  |
| 8. print (double d);   |                          |
| 9. print (boolean b);  |                          |

e.g.: import java.io.\*;

Class Test {

public static void main (String [7args]) {

FileWriter fw = new FileWriter ("abc.txt");

PrintWriter our = new PrintWriter (fw);

our.write(100); // 100 Unicode value will be written to file.

our.write ('\n');

our.println (100); // "100" will be written to the file

our.println (true);

our.println ("Denis");

our.flush();

our.close (); }

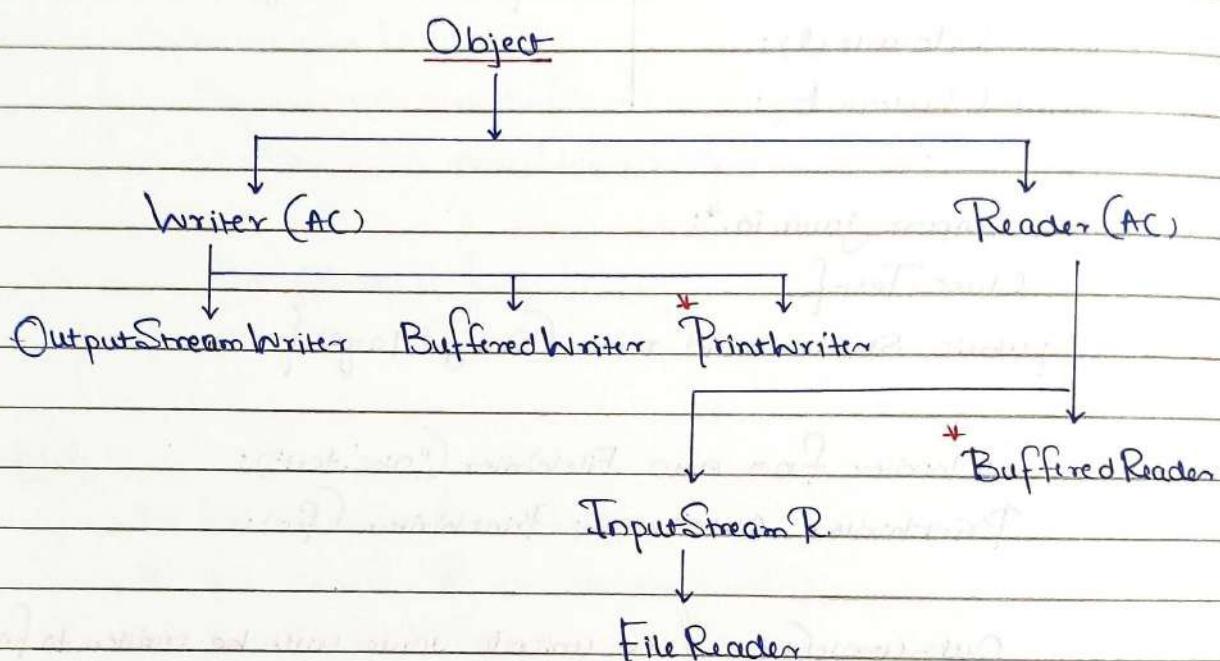
}

In case of print(100) '100' as it is  
will be written, but write(100)  
write the Unicode value 'd' to  
the file.

Note: The most enhanced Reader to read Character data from  
the file is "BufferedReader"

The most Enhanced Writer to write Character data to the File is  
"PrintWriter."

- \* In general we can use Readers and Writers to handle character data, whereas we use InputStreams and OutputStreams to handle binary data (like Images, audio file, video file etc).
  - \* We can use OutputStream to write binary data to the File and we can use InputStream to read binary data from the File.
- Character data → Reader and Writer  
 Binary data → InputStream and OutputStream



Program1: Copy all contents from file1.txt, file2.txt to file3.txt

```

import java.io.*;
class Test{
 public static void main (String [] args) throws Exception
 {
 PrintWriter pw = new PrintWriter ("file3.txt");
 // reading from first file and writing to file3.
 BufferedReader br = new BufferedReader (new FileReader
 ("file1.txt"));
 String line = br.readLine();
 }
}

```

```

 while (line != null)
 {
 pw.println(line);
 line = br.readLine();
 }
 // reading from second file and writing to file3
 br = new BufferedReader(new FileReader("file2.txt"));
 line = br.readLine();
 while (line != null)
 {
 pw.println(line);
 line = br.readLine();
 }
}

```

`pw.flush()` // to write all the data to file3.txt.

`br.close()`

`pw.close();`

`Sysout ("Open "file3.txt" to see the result"); }`

Program 2: Copy one line from file1.txt and from file2.txt to file3.txt.

```

 {
 PrintWriter pw = new PrintWriter("file3.txt");
 // reading from first file
 }

```

```

 BufferedReader br1 = new BufferedReader(new FileReader(
 ("file1.txt")));

```

`String line1 = br1.readLine();`

`// reading from second file`

```

 BufferedReader br2 = new BufferedReader(new FileReader("file2.txt"));

```

`String line2 = br2.readLine();`

```

 while (line1 != null || line2 != null)
 {
 if (line1 != null)
 {
 pw.println(line1);
 line1 = br1.readLine();
 }
 }
}

```

```

if (line2 != null)
{
 pw.println(line2);
 line2 = br2.readLine();
}
pw.flush(); // to write all data to file3.txt.
br1.close();
br2.close();
pw.close();
}

```

Program 3: W.A.P to remove duplicates from the file

```

{ BufferedReader br = new BufferedReader (new FileReader ("input.txt"));
PrintWriter pw = new PrintWriter ("output.txt");

String target = br.readLine();
while (target != null)
{
 boolean isAvailable = false;
 BufferedReader bri = new BufferedReader (new FileReader
 ("output.txt"));
 String line = bri.readLine();
 // Control comes out of loop in smooth fashion without break.
 while (line != null)
 {
 // if matched, control should come out of block with break
 if (line.equals (target))
 {
 isAvailable = true;
 break;
 }
 line = bri.readLine();
 }
 if (!isAvailable)
 pw.println(target);
}
br.close();
bri.close();
pw.close();
}

```

```

if (isAvailable == false)
{
 pw.println(target);
 pw.flush();
}

target = br.readLine(); }

br.close();
pw.close();
}

```

Program 4: Write a program to perform extraction of mobile no. only if there are no duplicates.

```

{ BufferedReader br = new BufferedReader (new FileReader
 ("input.txt"));
PrintWriter pw = new PrintWriter ("output.txt");
String target = br.readLine();

while (target != null)
{
 boolean isAvailable = false;
 BufferedReader bri = new BufferedReader (new FileReader
 ("dublicate.txt"));
 String line = bri.readLine();

 while (line != null)
 {
 if (line.equals(target))
 {
 isAvailable = true;
 break;
 }
 line = bri.readLine();
 }
 if (!isAvailable)
 pw.println(target);
}
br.close();
pw.close();
}

```

```
if (isAvailable == false)
{ pw.println (target);
pw.flush(); } // write all the data if false.
```

```
target = br.readLine(); }
```

```
br.close();
pw.close(); }
}
```

Program 5: W.A.P to read the data from file and identify which data is larger in length. (Data is string).

```
{ BufferedReader br = new BufferedReader (new FileReader
("data.txt"));
String data = br.readLine();
int maxLength = 0; String result = "";
while (data != null)
{
 int resLength = data.length();
 if (maxLength < resLength)
 {
 maxLength = resLength;
 result = data;
 }
 data = br.readLine();
}
```

```
System.out.println ("The max Length String is " + result);
```

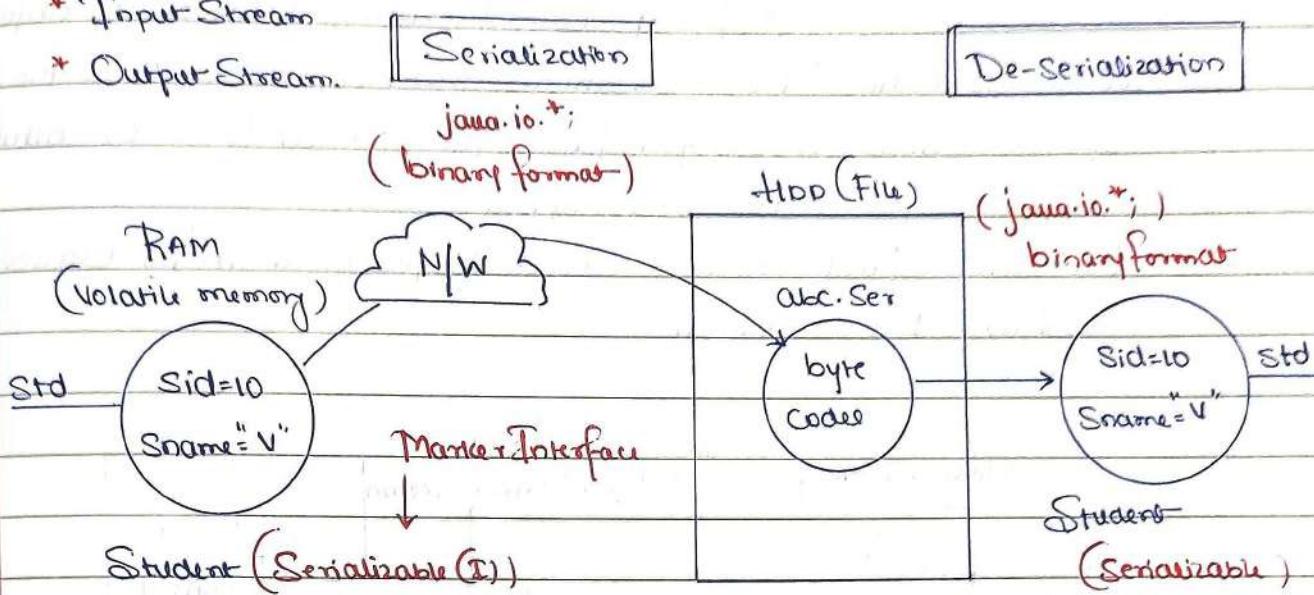
```
System.out.println ("The max Length of String is " + maxLength);
```

```
}
```

```
}
```

Binary Type of data.

- \* Input Stream
- \* Output Stream.

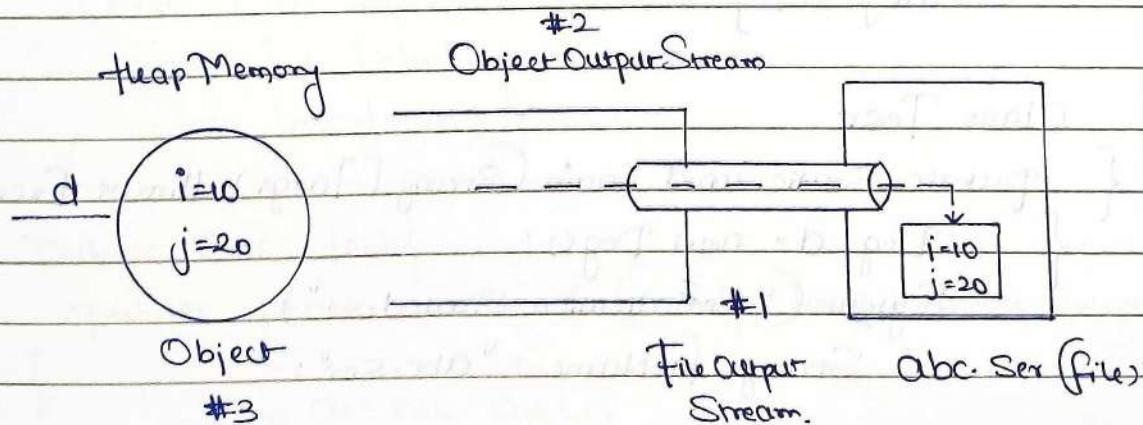


Key Points:

1. Object Should have the facility of transportation.
2. Object Should be Supported to Store inside file System.  
(Using Streams).

Serialization :

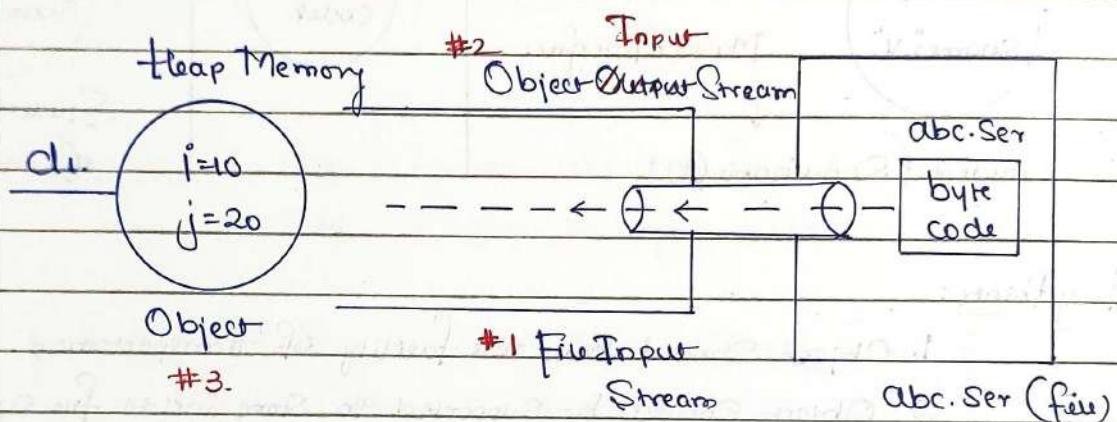
- \* The process of saving or writing state of an object to a file is called "serialization" but/or in other words it is a process of converting an object from "java supported form" to either network supported form or file supported form.
- \* By using "FileOutputStream" and "ObjectOutputStream" classes we can achieve serialization process.



## De-Serialization:

\* The process of reading state of an object from a file is called "De-Serialization", in other words it is the process of converting an object from file supported form or network supported form to java supported form.

\* By using "FileInputStream" and "ObjectInputStream" classes we can achieve De-serialization.



→ eg:: import java.io.\*;  
Class Dog implements Serializable {

    Static { System.out ("Static block gets executed"); }

    Dog() { System.out ("Object is created"); }

    int i=10;  
    int j=20; }

## Class Test

```

{ public static void main (String []args) throws Exception
{
 Dog d = new Dog();
 System.out ("Serialization Started... ");
 String fileName = "abc.ser";
 }

```

FileOutputStream fos = new FileOutputStream (fileName);

ObjectOutputStream oos = new ObjectOutputStream (fos);

oos. writeObject(d);

Sysout ("Serialized object reference is :" + d); // reference is

Sysout ("Serialization ended..."); "Dog@byk292"

// To pause the execution file we press some key from keyboard

System.in.read();

Sysout ("De-serialization started...");

FileInputStream fis = new FileInputStream ("abc.ser");

ObjectInputStream ois = new ObjectInputStream (fis);

Object obj = ois.readObject();

Dog d1 = (Dog) obj; // Object created but constructor not called

Sysout ("De-serialised object reference is :" + d1);

Sysout ("De-serialized process over"); } // reference is

} // Object creation happens internally "Dog@pk973"



eg:: import java.io.\*;

Class Dog implements Serializable {

int i=10;

int j=20; } // Class whose object to serialize must implement Serializable

Class Cat implements Serializable {

int i=100;

int j=200; }

Public class Test {

public static void main (String [ ] args) throws Exception

{ Dog d1 = new Dog ();

`Car c1 = new Car();`

`FileOutputStream fos = new FileOutputStream("abc.ser");`

`ObjectOutputStream oos = new ObjectOutputStream(fos);`

`oos.writeObject(d1); // the order in which serialized the`

`oos.writeObject(c1); Same order it should be de-serialized`

`FileInputStream fis = new FileInputStream("abc.ser");`

`ObjectInputStream ois = new ObjectInputStream(fis);`

`Dog d2 = (Dog) ois.readObject();`

`Car c2 = (Car) ois.readObject(); // order of deserialization`

}  
}      Should be maintained

### Note:

- \* We can perform serialization only for Serializable objects.
- \* An object is said to be Serializable if and only if the corresponding class implements Serializable interface.
- \* Serializable interface present in java.io package and does not contain any abstract methods. It is a marker interface, the required ability will be provided automatically by JVM.
- \* We can add any no. of Objects to the file and we can read all those objects from the file back in which order we wrote objects in the same order only the objects will come back.  
The order is important.
- \* If we try to serialize an object we get (nonSerializable object) then we get Runtime Exception : "NotSerializableException".

### Transient Keyword:

- \* transient is the modifier applicable only for variable, but not for classes and methods.
- \* While performing serialization if we don't want to save the value of a particular variable to meet security constraint such type

of variable, then we should declare it "transient".

- \* At the time of serialization JVM ignores the original value of transient variable and save default value to the file.
- \* That is - transient means "not to Serialize".

e.g.: `import java.io.*;`

```
Class Dog implements Serializable {
 int i = 10;
 transient int j = 20; }
```

Public class Test {

```
 Public static void main (String [] args) throws Exception
{
```

`Dog d1 = new Dog();`

`FileOutputStream fos = new FileOutputStream ("abc.ser");`

`ObjectOutputStream oos = new ObjectOutputStream (fos);`

`oos.writeObject (d1);`

`FileInputStream fis = new FileInputStream ("abc.ser");`

`ObjectInputStream ois = new ObjectInputStream (fis);`

`Dog d2 = (Dog) ois.readObject ();`

`Sysout (d2.i + " " + d2.j); } // 10 0`

Static vs transient: Static variable is not part of object state hence they won't participate in serialization because of this declaring a static variable as transient there is no use.

eg: {

`static transient int i=10;      Output : 10 20`

`int j=20; }`

(after serializ. and deserialization)

final v/s -transient: final variables will be participated into serialization directly by their values. Hence declaring a final variable as transient there is no use. (The compiler assigns the value to final variable).

eg:: final int x = 10;

int y = 20;

System.out.println(x); System.out.println(y);

eg:: { int i=10;  
transient final int j=20; } Output: 10 20

### Declaration output:

1. { int i=10; int j=20; } Output: 10 20
2. { transient int i=10; int j=20; } Output: 0 20
3. { transient int i=10; transient static int j=20; } Output: 0 20
4. { transient final int i=10; transient int j=20; } Output: 10 0
5. { transient final int i=10; transient static int j=20; } Output: 10 20

Note: We can serialize any number of objects to the file but in which order we serialized in the same order only we have to deserialize, if we change the order, it results in "ClassCast Exception".

eg:: { Dog d1 = new Dog();

Car c1 = new Car();

Rat r1 = new Rat();

FileOutputStream fos = new FileOutputStream("abc.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos. write Object (d);

oos. write Object (c);

oos. write Object (r);

// if we don't know the order of serialization then:

FileInputStream fis = new FileInputStream ("abc.ser");

ObjectInputStream ois = new ObjectInputStream (fis);

Object obj = ois.readObject();

if (obj instanceof Dog)

{ Dog d = (Dog) obj; }

if (obj instanceof Cat)

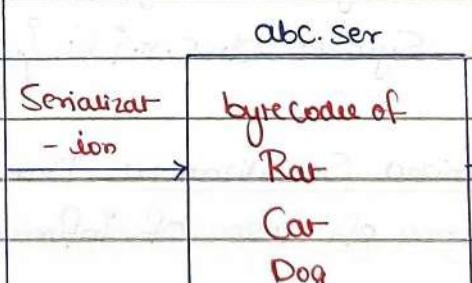
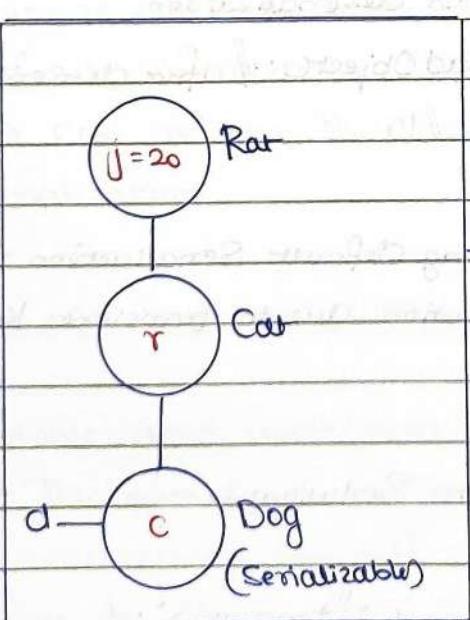
{ Cat c = (Cat) obj; }

if (obj instanceof Rat)

{ Rat r = (Rat) obj; }

}

### Object graph in Serialization



\* Refer next example.

### Object Graph

- Whenever we are serializing an object, the set of all objects which are reachable from that object will be serialized automatically. This group of objects is nothing but object graph in serialization.
- In the object graph every object should be Serializable. Otherwise we will get run-time exception saying "NotSerializableException".

eg:: Class Dog implements Serializable {  
    Car c = new Car(); }

Class Car implements Serializable {  
    Rat r = new Rat(); }

Class Rat implements Serializable {  
    int i = 10; }

Class Test {

```
public static void main (String [] args) throws Exception
{
 Dog d = new Dog();
 // Same dog code as previous example
 // Same Serialization and Deserialization.
 Dog d1 = (Dog) ois.readObject(); // after de-serialization.
 System.out.println(d1.c.r.i); // 10
}
```

Customized Serialization: During default serialization there may be a chance of loss of information due to transient keyword.

eg:: import java.util.\*;  
Class Account implements Serializable  
{  
    String name = "Sachin";  
    transient String password = "tendulkar"; }

```
public class Test{
```

```
 public static void main(String[] args) throws Exception
```

```
{
```

```
 Account acc = new Account();
```

```
 System.out.println(acc.name + " " + acc.password);
```

```
 FileOutputStream fos = new FileOutputStream("abc.ser");
```

```
 ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
 oos.writeObject(acc);
```

```
 FileInputStream fis = new FileInputStream("abc.ser");
```

```
 ObjectInputStream ois = new ObjectInputStream(fis);
```

```
 acc = (Account) ois.readObject();
```

```
 System.out.println(acc.name + " " + acc.password);
```

```
}
```

```
}
```

→ In the above code example before serialization Account object can provide proper username and password. But after De-serialization Account object can provide only Username but not password. This is due to declaring password as transient. Thus during default serialization there may be a chance of loss of info. We can reduce this loss of information by using customized serialization.

We can implement Customized Serialization by using 2 methods.

1. `private void writeObject(ObjectOutputStream os) throws Exception`

→ This method will be automatically executed by JVM at time of serialization. Hence if we want to perform any extra work we have to define that in this method only. (Prepare encrypted password and write enc. password separate to the file).

2. `private void readObject (ObjectInputStream ois) throws Exception;`  
 → This method will be automatically executed by Java at the time of de-serialization. Hence if we have to perform any extra work, we need to define that in this method only. (Read encrypted password, Perform decryption and assign decrypted password to the current object password variable.)

eg:: Class Account implements Serializable {

```
String name = "Sachin";
transient String password = "tendulkar";
```

```
private void writeObject (ObjectOutputStream oos) throws Exception
{
 oos. defaultWriteObject(); // performing default serialization
 String epwd = "123" + password; // encryption
 oos.writeObject(epwd); // write data to file (acc.ser)
}
```

```
public void readObject (ObjectInputStream ois) throws Exception
{
 ois.defaultReadObject(); // performing default serialization
 String epwd = (String) ois.readObject(); // decryption
 password = epwd.substring(3); } // writing data to object
}
```

public class Test {

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

```
Account acc = new Account();
```

```
System.out.println (acc.name + " " + acc.password);
```

```
FileOutputStream fos = new FileOutputStream ("abc.ser");
```

```
ObjectOutputStream oos = new ObjectOutputStream (fos);
```

```
oos.writeObject (acc);
```

FileInputStream fis = new FileInputStream ("abc.ser");

ObjectInputStream ois = new ObjectInputStream (fis);

acc = (Account) ois.readObject();

System.out.println(acc.name + " " + acc.password); }  
}

- At the time of Account Object Serialization JVM will check if there any writeObject() method in Account Class or not.
- If it is not available then JVM is responsible to perform serialization (default serialization).
- If Account Class Contain writeObject() method then JVM feels very happy and execute that Account Class.
- The same rule is applicable for readObject method also

e.g.: 2. To add another variable (transient) to previous example

{ transient int pin = 12345; // loss of information

public void writeObject (...) ... // changes to make in  
{ ..... previous example.

int encypin = 1111 + pin;

oos.writeInt (encypin); } // for int type data

public void readObject (...) ...

{ ..... int encypin = ois.readInt();

pin = encypin - 1111; }

}

### Serialization w.r.t Inheritance

Case 1: If parent class implements Serializable then automatically every Child class by default implements Serializable

That is Serializable nature is inheriting from parent to child. Hence even though Child class doesn't implement Serializable, we can Serialize Child class object if parent class implements Serializable interface.

eg:: Class Animal implements Serializable {  
    int i=10; }

Class Dog extends Animal {  
    int j=20; }

public class Test {  
    public static void main(String[] args)  
    {  
        Dog d = new Dog();  
        // Serialization process..

// DeSerialization process (Same as in previous Dog example)

Dog d1 = (Dog) ois.readObject();  
System.out.println(d1.i + " " + d1.j); } // 10 20  
}

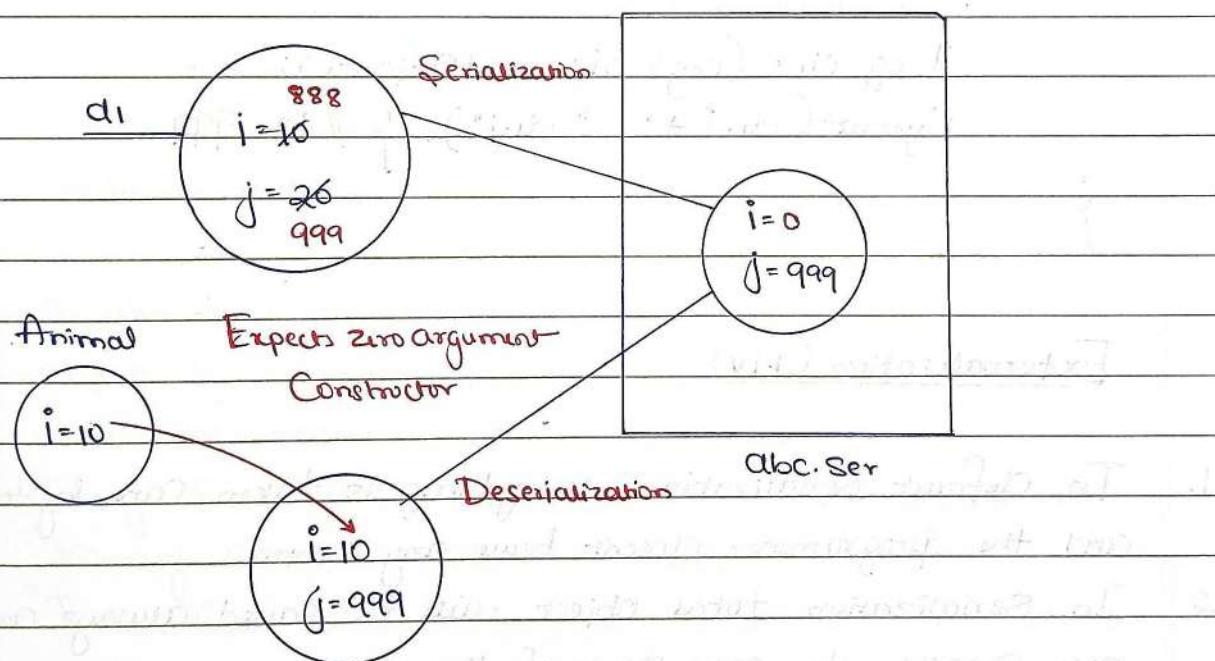
- \* Even though Dog class does not implement Serializable explicitly but we can Serialize Dog object because its parent class already implements Serializable interface.
- \* Object class doesn't implement Serializable interface.

### Case 2:

- \* Even though parent class does not implement Serializable we can Serialize child object if Child class implements Serializable interface.
- \* At the time of serialization Jvm ignores the value of instance

variable which are coming from non Serializable parent then instead of original value JVM saves default value for those variable to the file.

- \* At the time of de serialization JVM checks whether any parent class is non Serializable or not. If any parent class is non Serializable JVM creates a separate object for every non Serializable parent and shares its instance variable to the current object.
- \* To create an object for non-Serializable parent hence JVM always calls no argument constructor of that non-Serializable parent hence every parent should contain no-arg constructor otherwise we will get runtime exception "InvalidClassException".



eg:: import java.io.\*;

```
Class Animal{
```

```
 int i=10;
```

```
 Animal(){ System.out.println("no arg Animal constructor");}
```

```
}
```

Class Dog extends Animal implements Serializable {  
 int j=20;

```
Dog(); System.out.println("no arg Dog constructor");
}
```

```
public class Test{
 public static void main(String[] args) throws Exception
 {
 Dog d = new Dog();
 d.i = 888;
 d.j = 999;
```

// Serialization takes place like previous example.

// De-Serialization happens

```
Dog d1 = (Dog) ois.readObject();
System.out.println(d1.i + " " + d1.j); // 10 999
}
```

## Externalization (1.1v)

1. In default Serialization everything is taken care by the JVM and the programmer doesn't have any control.
2. In Serialization total object will be saved always and it is not possible to save part of the object, which creates performance problems at certain points.
3. To overcome these problems we should go for externalization where everything is taken care by the programmer and JVM doesn't have any control.
4. The main advantage of externalization over serialization is we can save either total object or part of the object based on our requirement.

5. To provide externalizable ability for any object compulsory the corresponding class should implements Externalizable interface.
6. Externalizable interface is child interface of Serializable interface.

Externalizable interface defines 2 methods:

1. `public void writeExternal(ObjectOutput out) throws IOException`  
This method will execute automatically at the time of serialization with this method, we have to write code to save required variable to the file.
  2. `public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException`  
This method will be executed automatically at the time of de-serialization with this method we have to write code to save read required variable from file and assign to the current object.
- At the time of deserialization Jvm will create a separate new object by executing Public no-arg constructor on that object Jvm will call `readExternal()` method.
- Every Externalizable class should compulsorily contain public no-arg constructor otherwise we get Runtime exception saying "Invalid Class Exception".

eg:: `import java.io.*;`  
Class Demo implements Externalizable {

    String i;

    int k;

    int j;

    Demo (String i, int k, int j) {

        this.i = i;

        this.j = j;

```
-this.k = k; }
```

```
public Demo()
{
 System.out.println("zero arg constructor"); }
```

### // Performing Serialization as required

```
public void writeExternal(ObjectOutput out) throws IOException
{
 System.out.println("call back method used while serialization");
 out.writeObject(i); // only required object are
 out.writeObject(j); } serialized
```

### // Performing De-serialization as per requirement

```
public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
```

```
System.out.println("call back method used while de-serialization");
i = (String) in.readObject();
j = in.readInt(); }
```

```
public class Test{
```

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

```
Demo d = new Demo("nitin", 100, 200);
```

// Serialization Started

```
FileOutputStream fos = new FileOutputStream("abc.ser");
```

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
oos.writeObject(d);
```

### // De-serialization Started

```
FileInputStream fis = new FileInputStream("abc.ser");
```

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

```
d = (Demo) ois.readObject();
```

```
System.out.println(d.i + " " + d.j + " " + d.k); }
```

Output:

Serialization Started

Callback method used while Serialization

Serialization ended

De-serialization Started

Zero-arg Constructor

Call back method used while De-serialization

nitin 100 0

Deserialization ended.

- If the Class implements Externalizable interface then only part of the object will be saved
- If the class implements Serializable interface then the output is "nitin 100 200".
- In externalization transient keyword won't play any role, hence transient keyword not required

| Serialization                                                             | Externalization                                                              |
|---------------------------------------------------------------------------|------------------------------------------------------------------------------|
| 1. It is meant for default serialization                                  | It is meant for customized serialization                                     |
| 2. Everything is taken care by JVM, programmer don't have control.        | Everything is taken care by programmer<br>JVM does not have any control -    |
| 3. Total object will be saved always.                                     | We can save total object or partial based on our requirement.                |
| 4. Relatively performance is low                                          | Performance is high.                                                         |
| 5. Serializable (T) doesn't contain methods.                              | Externalizable contains 2 methods (I).                                       |
| 6. It is a marker interface                                               | It is not a marker interface                                                 |
| 7. Serializable class not necessary to contain public no-arg constructor. | Externalizable class should compulsorily contain public no-arg constructor - |
| 8. transient keyword play role in serialization.                          | - transient keyword don't play any role in externalization                   |

### SerialVersionUID :

- \* To perform Serialization and De-Serialization internally JVM will use a unique identifier, which is serialVersionUID.
- \* At the time of serialization JVM will save serialVersionUID with object.
- \* At the time of de-serialization JVM will compare serialVersionUID and if it matches then only de-serialization can be done or else results in runtime exception saying "InvalidClassException".

Process In depending in Default serialVersionUID are:

- After serializing object if we change the .class file then we can't perform de-serialization because of mismatch in serialVersionUID of local class and serialized object in this case at the time of de-serialization we will get runtime exception saying "Invalid ClassException".
- Both sender and receiver should use the same version of JVM if there is any incompatibility in JVM versions then receiver will be unable to deserialize because of different serialVersionUID, in this case receiver will get "InvalidClassException".
- To generate serialVersionUID internally JVM will use complex algorithm which may create performance problems.

We can solve above problem by Configuring our own serialVersionUID  
eg::

```
import java.io.Serializable;
public class Dog implements Serializable {
 private static final long serialVersionUID = 1L;
 int i=10;
 int j=20; }
```

Sender

Class Dog implements Serializable

{  
    private final static long

SerialVersionUID UID = qL;

int i=10;

int j=20;

}

i=10  
j=20

TND

Serialization

Windows

Jdk 1.6 (Oracle)

Serialization



Receiver

Class Dog implements Serializable

{  
    private final static long

SerialVersionUID UID = qL;

int i=10;

i=10  
j=20

j=20

OK

De-Serialization

MAC

JDK 1.6 (IBM)

abc.ser

De  
Serialization.

Bytecode  
OP  
i=10  
j=20

SerialVersionUID UID = qL

```

import java.io.*;
public class Sender{
 public static void main (String [] args) throws IOException {
 Dog d = new Dog();
 FileOutputStream fos = new FileOutputStream ("abc.ser");
 ObjectOutputStream oos = new ObjectOutputStream (fos);
 oos.writeObject (d);
 }
}

```

```

import java.io.*;
public class Receiver{
 public static void main (String [] args) throws IOException,
 ClassNotFoundException {
 FileInputStream fis = new FileInputStream ("abc.ser");
 ObjectInputStream ois = new ObjectInputStream (fis);
 Dog d2 = (Dog) ois.readObject();
 System.out.println (d2.i + " " + d2.j);
 }
}

```

### Commande:

```

javac Dog.java
java Sender
javac Dog.java
java Receiver. Output: 10 20

```

- In the above programs after serialization even though if we perform any change to Dog class file we can de-serialize object.
- We can configure our own serialVersionUID, both sender and receiver not required to maintain the same JVM version.
- Some IDE's generate explicit serialVersionUID

## String Tokenizer

- \* It is a part of java.util package
- \* It is used to split the entire string into multiple tokens based on the delimiter we supply.

eg: String data = "Sachin ramesh tendulkar";

StringTokenizer str = new StringTokenizer(data);

// splits the string at empty Space

StringTokenizer str = new StringTokenizer(data, "@");

// splits at '@'.

- \* Other Methods:

→ public boolean hasMoreTokens();

→ public String nextToken();

eg:: import java.util.\*;

Class Test{

public static void main(String []args)

{

StringTokenizer str = new StringTokenizer("Sachin \$ ramesh \$ tendulkar", "\$");

System.out.println(str);

int tokens = str.CountTokens();

System.out.println(tokens);

while (str.hasMoreTokens())

{ String data = str.nextToken();

System.out.println(data); }

}

}

Output: StringTokenizer @7ash2q

3

Sachin

ramesh

tendulkar