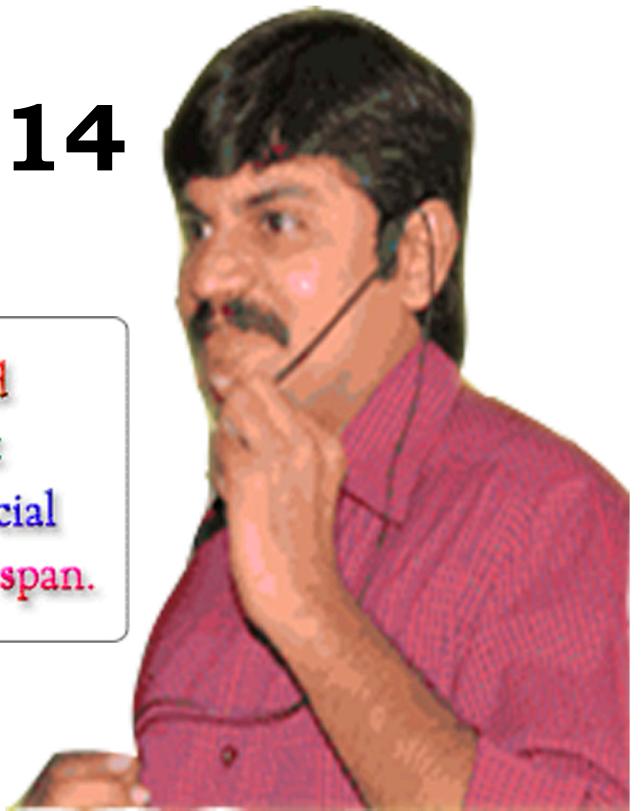


# **SC.JP NOTES**

## **NEW NOTES 2014**



Our dedication and  
students full effort  
makes us very special  
with in very less span.



**SRINIVASA T - INFOTECH SOLUTIONS**



**SHOP NO - 1 , SJRJ TOWERS ,  
BESIDE PRIME HOSPITAL LANE ,  
AMEERPET , HYDERABAD ,**

**PH NO : 9618475707 ,  
7794958589.**



031-07-2014

## Scjp

maelmu.13

1. Language Fundamentals 1-56
2. Declaration and Access Control 99-145
3. Operators & Assignments 57-77
4. Flow Control 78-98
5. Exception Handling 147-186
6. Assertions 187-194
7. Garbage Collection 195-204
8. OOPS 205-258
9. Multi-Threading 259-
10. java.lang. package - Object  
String, StringTokenizer, StringBuilder  
Wrapper Classes, Autoboxing
11. java.io. Package
12. Serialization
13. Collection Framework
14. Generics
15. Inner classes
16. Internationalization (I18n)
17. Development
18. Regular Expressions
19. enum
20. JVM Architecture

01/Aug/2014

## Language Fundamentals

- ① Identifiers
- ② Reserved words
- ③ Datatypes
- ④ Literals
- ⑤ Arrays
- ⑥ Types of Variables
- ⑦ var-arg methods
- ⑧ main method
- ⑨ Command-line arguments
- ⑩ Java Coding Standards

### ① Identifiers

A name in Java program is called Identifier which can be used for identification purpose. It can be class name or method name or variable name or label name.

Class Testd  
public static void main(String [] args)  
{  
    int x=10;  
}  
// 5 Identifiers

Rules for Java Identifiers

Rule ① → The only allowed characters in Java Identifiers are

a to z  
A to Z  
0 to 9  
\$  
\_ (underscore)

If we are using any other character then we will get Compile time error

Eg: total-number ✓  
total # X Compile time error

Rule ② Identifier can not start with digit

Eg: total123 ✓ valid  
123 total X invalid

Rule ③ Java identifiers are case sensitive of course Java language itself considered as case sensitive programming language.

(3)

Eg:

```

class Test
{
    int number=10;
    int Number=20;
    int NUMBER=30;
}

```

We can differentiate with respect to case

Rule (4) There is no length limit for java identifiers but it's not recommended to take too lengthy identifiers.

Rule (5) we can't use Reserved words as identifiers otherwise we will get compilation error

Eg: int x=10; ✓  
int if = 20; X

Rule (6) All predefined java class names and interface name we can use as identifiers

```

class Test {
    public static void main(String[] args)
    {
        int String = 888;           → class
        System.out.println("Hello World"); |   int Runnable = 999;           → interface
        System.out.println("Hello World"); |   System.out.println(Runnable);
    }
}

```

→ Even though it is valid to use predefined java class names and interface names as identifiers but it is not a good programming practice.

which of the following are valid java identifiers?

total\_number ✓      total123 ✓

total# X      123total X

all@hands X

java2share ✓

Cash ✓

int X (reserved word)

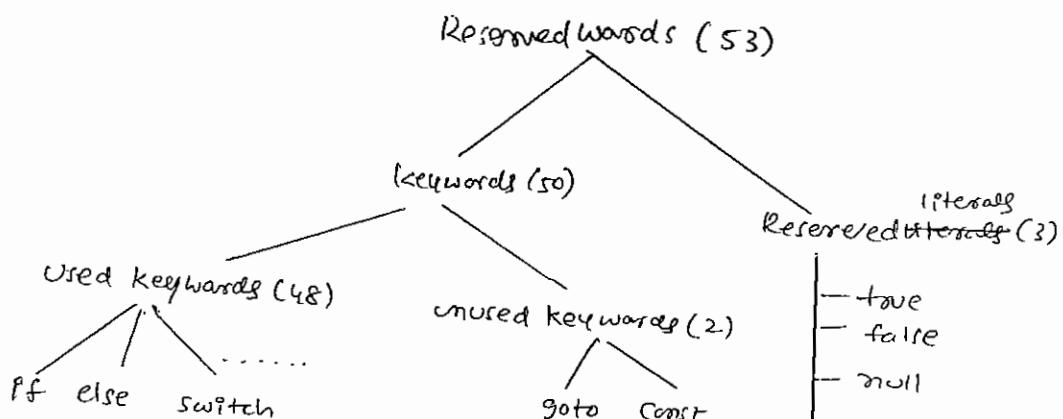
Int ✓

Integer ✓ (Predefined class name)

-\$-\$ ✓

## ② Reserved words

In Java some words are reserved to represent some meaning or functionality such a type of words are called reserved words.



### Reserved words for Data Types ⑧

- byte
- short
- int
- long
- float
- double
- boolean
- char

### Reserved words for flow control ⑪

- if
- else
- switch
- case
- default
- while
- do
- for
- break
- continue
- return

### Reserved words for modifiers ⑪

- public
- private
- protected
- final
- abstract
- static
- synchronized
- native
- Strictfp (1.2v)
- transient
- volatile

### Reserved words for Exception handling ⑥

- try
- catch
- finally
- throw
- throws
- assert (1.4v) // used for debugging purpose

### Class related keywords ⑥

- class
- interface
- package
- import
- extends
- implements

### Object related keywords ④

- new
- instanceof
- super
- this

### void return type keyword ①

→ In Java return type is mandatory if a method won't return anything such type of methods we should declare with void return type.

Note: In 'C' Language return type is optional and default return type of int

### Un Used Keywords (2)

1. goto: Usage of goto Keyword created several problems in old languages like C or C++ hence SUN people ban this keyword in java.
2. Const: use final instead of const.

Note: goto and const are unused keywords if we are trying to use them we will get compilation error.

### Reserved Literals (3)

true	}	values for boolean datatype
false	}	
NULL	}	default value for Object reference.

### enum (1.5v)

We can use enum to define a group of named constants

```
Ex: enum Month
{
    JAN, FEB, ..., DEC;
}

enum Beer
{
    KF, KO, RC, FO;
}
```

### Conclusion 8

- 1) All 53 reserved words in java contain only lowercase alphabet symbols
- 2) In java we have only new keyword but not delete because destruction of useless objects is the responsibility of garbage collector.
- 3) The new keywords in java are

Strictfp — 1.2v  
assert — 1.4v  
enum — 1.5v

- 4) Const but not Constant  
instanceof but not instanceof  
Strictfp but not StrictFP  
synchronized but not synchronize  
implements but not implement  
extends but not extend  
import but not imports

Which of the following list contains only Java Reserved words

new, delete

goto, constant

break, continue, return, exit

final, finally, finalize → methods

throw, throws, thrown

notify, notifyAll

implements, extends, compo<sup>x</sup>nents

sizeof, instanceof

instanceof, sizeof

byte, short, <sup>x</sup>int

None of these

Which of the following are valid Java Reserved words

public ✓

static ✓

void ✓

main

String

args

monday  
② [04-08-2014]

### ③ Data Types

1) In Java every variable and every expression should have some type and every type is strictly defined.

Each and every assignment should be checked by the compiler for type compatibility. Hence Java language is considered as strongly typed programming language.

Eg: int x = 10; ✓  
      int x = 10.5; <sup>(x)</sup> C E X

2) Java is not considered as pure Object Oriented Programming language because several OOP features (like operator overloading, multiple inheritance etc.) are not supported by Java.

Moreover we are using primitive data types which are non objects

#### Primitive Data Types (8)

##### Numeric data types

##### Non-Numeric data types

###### Integral data types

byte  
short  
int  
long

Eg: 100

###### Floating-Point data types

float

double

boolean

char

Eg: 100.234

(6)

→ Except boolean and char remaining datatypes are considered as signed data types because we can represent both the positive and negative numbers.

Ex:

 $\text{int } x = +10; \checkmark$   
 $-10; \checkmark$ 
 $\text{double } d = +10.5; \checkmark$   
 $-10.5; \checkmark$ 

signed

 $\text{char } z = '@' \checkmark$ 
 $\text{boolean } = -\text{true}; \times$   
 $\text{false}; \times$ 

unsigned

byte b

Size: 1 byte (8 bits)

max\_value : ~~+128~~ +127

min\_value : -128

Range : -128 to +127

MSB

X	1	1	1	1	1	1	1	1
	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
	$64 + 32 + 16 + 8 + 4 + 2 + 1$							

Sign bit

 $0 \rightarrow \text{true}$  $1 \rightarrow \text{false}$ byte b = -128;  $\checkmark$ 

→ The most significant bit acts as sign bit

0 → means +ve number

1 → means -ve number

→ Positive numbers will be represented directly in the memory whereas as negative numbers will be represented in 2's complement form.

Ex:

 $\checkmark \text{byte } b = 10;$  $\checkmark \text{byte } b = 127;$  $\times \text{byte } b = 128;$ 

C.E : Possible loss of precision  
 found: int  
 required: byte

C.E compile time error

 $\times \text{byte } b = 10.5;$ 

C.E : Possible loss of precision  
 found: double  
 required: byte

 $\times \text{byte } b = \text{true};$ 

C.E Incompatible types  
 found: boolean  
 required: ~~byte~~ byte

 $\times \text{byte } b = "durga";$ 

C.E Incompatible types  
 found: java.lang.String  
 required: byte

byte data type is best suitable to handle data in terms of streams either from the file or from the network.

### Short

1) Short is the most rarely used datatype in java.

Size : 2 bytes (16 bits)

Range :  $-2^{15}$  to  $2^{15}-1$

[ -32768 to 32767 ]

Ex:

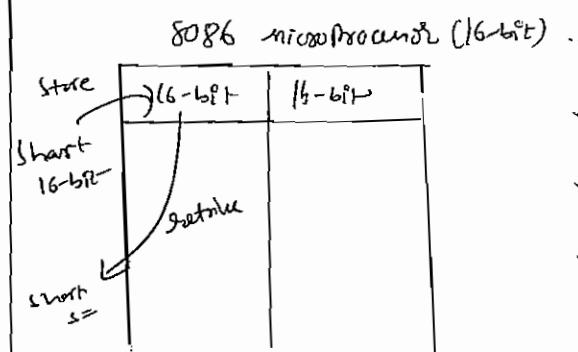
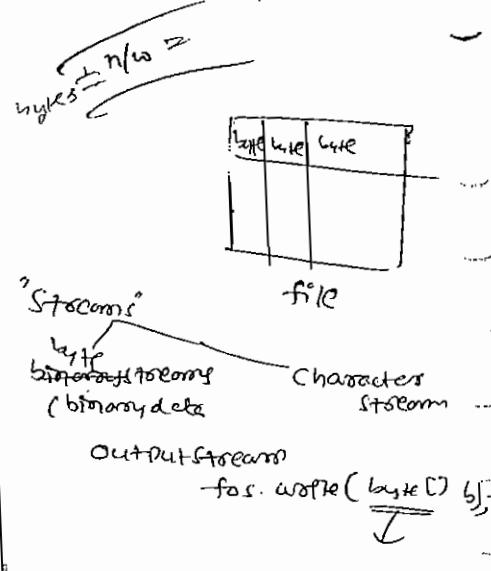
Short s = 10; ✓

X Short s = 32768; { C.E: Possible loss of precision  
found: int  
required: short

X Short s = 10.5; { C.E: Possible loss of precision  
found: double  
required: short

X Short s = false; { C.E incompatible types  
found: boolean  
required: short

→ Short data type is best suitable to handle for 16-bit processors like 8086 but these processors are outdated and hence corresponding short datatype is also outdated datatype.



int

- It is the most commonly used datatype in Java

Size : 4 bytes (32 bits)

Range is  $-2^{31}$  to  $2^{31}-1$

$$[-2147483648 \text{ to } 2147483647]$$

Eg: `int x = 2147483647;`

~~X int x = 2147483648;~~

CE: Integer number too large

~~↳ int x = 2147483648 l;~~

CE: Possible loss of precision

found: long

required: int

~~X int x = true;~~

CE: Incompatible types

found: boolean

required: int

long

Some time int may not enough to hold big values then we should go for long datatype.

1000 days

Eg: 1) To hold the amount of distance travelled by light in ~~the universe~~ 1000 days  
int may not enough we should go for long type.

`long l = 1,26,000 * 60 * 60 * 24 * 1000 miles;`  
 $\downarrow$   
 day

Eg 2:)

To hold the number of characters present in a big file int may not enough we should go for long type hence the datatype or length() method is long

`long l = f.length();`

Size: 8 bytes [64 bits]

Range is  $-2^{63}$  to  $2^{63}-1$

contains → number of pages  
 → number of lines  
 → number of characters

file

Notes (05-08-2014)

Note & all the above datatypes meant for representing integer values

↳ If we want to represent floating point values then we should go for floating point datatypes.

## Floating Point datatypes

### Floating-Point datatype

float

double

- 1) If we want 5 to 6 decimal places of accuracy then we should go for float
- 2) float follows single precision  
(less accuracy)
- 3) Size: 4 bytes
- 4) Range:  $-3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$   
 $-3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$
- 5) If we want 14 to 15 decimal places of accuracy then we should go for double.
- 6) double follows double precision  
(more accuracy)
- 7) Size: 8 bytes
- 8) Range:  $-1.7 \times 10^{-308}$  to  $1.7 \times 10^{308}$   
 $-1.7 \times 10^{-308}$  to  $1.7 \times 10^{308}$

## boolean datatype

Size: Not applicable ( JVM dependent )

Range: Not applicable ( but allowed values are true or false )

e.g:

✓ boolean b = true;

✗ boolean b = 0;      CSE: Incompatible types

✗ boolean b = True;

found: int  
Required: boolean

not valid value for  
any datatype  
Compatibility by defining  
true or false variable

✗ boolean b = "true"

CSE  
Can not find symbol  
Symbol: Variable True  
Location: class Test

CSE  
Incompatible types

found: java.lang.String  
Required: boolean

```
int x=0;
```

```
if(x)
```

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

```
else
```

```
{ sopen("Hi"); }
```

```
}
```

↳ If always expect boolean

while - always expect boolean

↳ incompatible types

found : int

required : boolean

while (i)

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

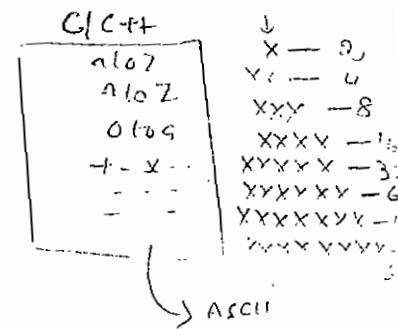
```
}
```

1 bit = 2 values  
0 or 1

Y-bit

## Char datatype

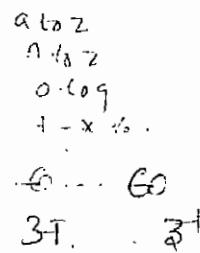
- old languages (like C/C++) are ASCII based and the number of ASCII characters are less than or equal to 256 ( $\leq 256$ ) to represent these 256 characters 8 bits are enough hence size of char in old languages is 1 byte.



- but Java is Unicode based and the number of Unicode characters are  $> 256$  and  $\leq 65536$  hence to represent these many characters 8 bits are not enough compulsory we should go for 16 bits. due to this the size of char in Java is 2 bytes

Size: 2 bytes

Range: 0 to 65535 ( $0 + 65535 = 65536$ ) (16-bits)



## Summary of Java primitive data types

UNICODE

datatype	size	Range	Corresponding wrapper class	default value
byte	1 byte	$-2^7$ to $2^7 - 1$ (-128 to 127)	Byte	0
short	2 bytes	$-2^{15}$ to $2^{15} - 1$ (-32768 to 32767)	Short	0
int	4 bytes	$-2^{31}$ to $2^{31} - 1$ (-2147483648 to 2147483647)	Integer	0
long	8 bytes	$-2^{63}$ to $2^{63} - 1$	Long	0
float	4 bytes	$-3.4 \times 10^{-38}$ to $3.4 \times 10^{-38}$	Float	0.0
double	8 bytes	$-1.7 \times 10^{-308}$ to $1.7 \times 10^{-308}$	Double	0.0
boolean	not applicable	not applicable (But allowed values are true/false)	Boolean	False
char	2 bytes	0 to 65535	Character	→ new space char

null

null is the default value for Object reference and we can't apply

for primitive types

If we are trying to assign null value for primitive datatypes then we will get compiletime error

Eg:

```
char ch = null;
```

C:\> Incompatible types

found : <null type>

required : char

## ④ Literals

Any constant value which can be assigned to the variable is called Literal

Eg: int x=10;

datatype / keyword

name of variable / identifier

constant value / literal

### Integral Literals

for integral datatypes (byte, short, int, long) we can specify literal value in the following ways

#### 1) decimal literals

allowed digits are 0 to 9

Eg: int x=10;

#### 2) octal literals (base - 8)

allowed digits are 0 to 7

literal value should be prefixed with zero (0)

Eg: int x= 010;

#### 3) hexadecimal literals (base - 16)

allowed digits are 0 to 9, a to f

for extra digit (a to f) we can use both lowercase and uppercase characters.

This is one of very few areas where java is not case sensitive.

literal value should be prefixed with zero (0X) or (0x)

Eg: int x= 0X10;    int x=0x10;



- We can't specify byte & short literals directly whenever we are assigning integral literal to the byte variable and if the value will in the range of byte then compiler automatically treats it as byte literal similarly short literal also.

Eg: ✓ byte b = 10;

✓ byte b = 127;

✗ byte b = 128;

C-E  
Possible loss or precision

found: int  
required: byte

short s = 32767;

✗ short s = 32768;

C-E

Possible loss or Precision  
found: int  
required: short

### Floating point literals

- By default every floating point literal is of double type and hence we can't assign directly to float variable.
- but we can specify floating point literal explicitly as float type by suffixed with f (or) F.

Eg:

✗ float f = 123.456;

✓ float f = 123.456f;

✓ double d = 123.456;

✓ double d = 123.456f;

C-E

Possible loss or precision  
found: double  
required: float

- ↳ but we can specify explicitly floating point literal as double type by suffixed with d or D Of course this conversion is not required

Eg:

double d = 123.456D;

- \* → we can specify floating point literals only in decimal form.
- octal and hexa decimal forms are not allowed.

Eg: double d = 123.456;

double d = 0123.456;

{ at is not treated as  
octal if it is treated as decimal

X double d = 0x123.456;

←← malformed floating point literals

found d = 0x123456

→ We can assign integral literal directly to the floating point variable and that integral literal can be specified either in decimal or octal or hexadecimal form.

double d = 0786;

octal ↗

double d = 10,  
for d = 10.0

Ex:

— X double d = 0786; C.E. (integer number too large)

✓ double d = 0xFace;

✓ double d = 0777;

X double d = 0xFace.0;

✓ double d = 0786.0;

\* → but we can't assign floating point literals directly to the integral types ↗ C.E.

Ex: int x = 10.5; X C.E. (possible loss of precision)  
found double required int

\* → We can specify floating point literal even in exponential form also (also known as scientific notation)

Ex: double d = 1.2e3

Sop(d); // 1200.0

1.2e3  
1.2 × 10<sup>3</sup> ⇒  
1.2 × 1000  
1200.0

X float f = 1.2e3; → C.E. Possible loss of precision  
found double required float

✓ float f = 1.2e3F;

## boolean literals

• The only allowed values for boolean datatype are true (or) false where case should be in lowercase.

Ex: ✓ boolean b = true;

X boolean b = 0; → C.E.:

X boolean b = True;

Incompatible types  
found: int  
required: boolean

X boolean b = "true"; ↗

C.E.:

Incompatible types

found: java.lang.String

required: boolean

C.E.: cannot find symbol  
symbol: Variable True  
location: class Test

int x=0;  
 if (x)  
 {  
 System.out.println("Hello");  
 } else  
 {  
 System.out.println("Hi");  
 }

C.E   Incompatible types

found: int  
 required: boolean

while (1)  
 {  
 System.out.println("Hello");  
 }

All these If, While expect only boolean type  
Enter true (or) false.

### Char Literals

- A char literal can be specified as single character with in single quotes

Eg. char ch='a'; ✓

X char ch="a"; ↗ C.E

Incompatible types  
found: java.lang.String  
required: char

X char ch=a; ↗ C.E

(can not find symbol)  
symbol: variable a  
location: class Test

X char ch='ab';

C.E<sup>1</sup> Enclosed Character Literal

char ch='ab';

C.E<sup>2</sup> Unclosed Character Literal

char ch='ab' ;

Value      0xface  
↳      64502

C.E<sup>3</sup> not a statement

char ch='ab';

- We can specify char literal as integral literal which represents UNICODE value of that character.
- The integral literal can be specified either in decimal or octal or hexadecimal forms.

The allowed range is 0 to 65535

Eg: char ch='a'; ✓

System.out.println(ch); ↗ a

✓ char ch=65535;

✓ char ch=65536;

C.E Possible loss of precision

↳ found: int

required: char

char ch=0xFaces; ✓

char ch=0777; ✓

char ch=0xBEEF; ✓

char ch=0xBEEF; X

char ch=0786; X

Ques 108/14)

• Unicode representation of char literal:

- we can represent char literal even in unicode representation which is nothing but

e \uXXXX'

↓ 4-digit hexadecimal number

Eg:

char ch = '\u0061';

Sop(ch); // a ✓

char ch = '\uface'; ✓

X char ch = '\iface';

X char ch = '\ubeef';

- Every escape character in java is a char literal

Eg:

char ch = '\n';

char ch = '\t';

X char ch = '\m'; C.E: illegal escape character

#### Escape Character

#### Description

\n

new line character

\t

Horizontal tab

\r

Carriage return

\b

Backspace character

\f

Form feed

\'

Single quote

\"

Double quote

\|

Black slash

## String literals

- A sequence of characters within " " (double quotes) is considered as string literals

Eg: String s = "durga"; ✓

## 1.7 Enhancements with respect to literals

### Binary literals

for integral datatypes (byte, short, int, long) until 1.6 version we can specify literal value in the following ways

- i) decimal literal
- ii) octal literal
- iii) hexadecimal literal

- But from 1.7 version onwards we can specify literal value even in binary form also. The allowed digits are  $\underbrace{0 \text{ or } 1}_{\text{0b (or) OB}}$  zero
  - Literal value should be ~~prefixed~~ prefaced with  $\text{0b}$  or  $\text{OB}$
- Eg: `int x = 0B1111;`  
`sop(x);`  
`O/P:`

"Case of - (underscore) b/w digits in numeric literals:

- From 1.7 onwards we can use  $_$  symbol between digits of numeric literals

Eg: `double d = 1_23_456.7_8_9;`

`double d = 1_23_456.7_8_9;`

`double d = 123_456.7_8_9;`

- The main advantage of this approach is readability of the code will be improved.
- At the time of compilation compiler will remove these  $_$  (underscore) symbol automatically after compilation the above lines will become

`double d = 123456.789;`

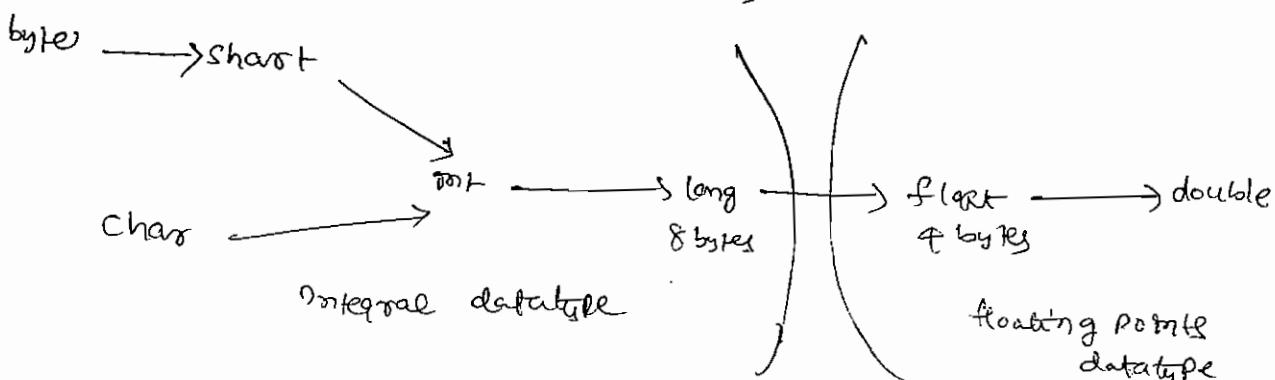
- We can use more than one  $_$  symbols also b/w the digits

`double d = 1__23__4__5__6.7__8__9;`

`double d = 1__23__4__5__6__7__8__9;`

- We can use  $_$  symbol only b/w the digits otherwise we will get compile time error

X {   
~~double d = 123456.789;~~  $= 1_23_456.7_8_9;$   
~~double d = 1-23-456.7-8-9;~~  
~~double d = 1-23-456.7-8-9;~~  
~~double d = 1-23-456.7-8-9;~~



Note: 8 byte long value can be assigned to a 4 byte float variable because both are following different memory representation

Eg: `float f = 10.0;` ✓  
Solved → 110.0

## Arrays

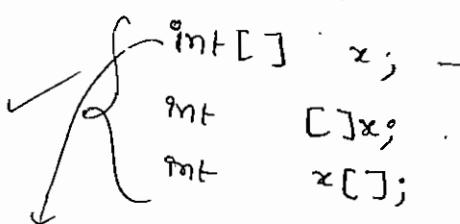
- ① Introduction
- ② Array declaration
- ③ Array creation
- ④ Array initialization
- ⑤ Array declaration, creation, initialization in a single line
- ⑥ length vs length()
- ⑦ Anonymous Arrays
- ⑧ Array element assignments
- ⑨ Array variable assignments

### ① Introduction

- An Array is an indexed collection of finite number of homogeneous data elements
- The main advantage of Arrays is we can represent multiple values by using single variable, so that readability of the code will be improved.
- But the main disadvantage of the arrays is fixed in size.  
I.e once we creates an array with some size there is no chance of increasing or decreasing the size based on our requirement. Hence to use arrays compulsorily we should know the size in advance which may not possible always.

### ② Array declaration

One-dimensional array declaration

  
`int[] x;` → Recommended only  
`int x[];`  
`int x[ ];`

Note: Recommended because name is clearly separated from type.

At the time of array declaration we can't specify the size otherwise we will get compilation error.

Ex:

~~`x int[6] x;`~~  
`int[] x;`

## Two dimensional array declaration

✓  $\left\{ \begin{array}{l} \text{int } [][] x; \\ \text{int } - \quad [][]x; \\ \text{int } \quad x[][],; \end{array} \right.$   
 ✓  $\left\{ \begin{array}{l} \text{int } [] \quad []x; \\ \text{int } [] \quad x[]; \\ \text{int } \quad []x[]; \end{array} \right.$

## Three dimensional array declaration

✓  $\text{int } [][][] x;$   
 ✓  $\text{int } \quad [][][]x;$   
 ✓  $\text{int } \quad x[][][];$   
 ✓  $\text{int } [] \quad [][]x;$   
 ✓  $\text{int } [] \quad []x[];$   
 ✓  $\text{int } \quad []x[][],;$   
 ✓  $\text{int } [] \quad x[][],;$   
 ✓  $\text{int } \quad x[][],;$

Which of the following declarations are valid

✓  $\text{int } [ ] a, b; \quad \begin{matrix} a \rightarrow 1 \\ b \rightarrow 1 \end{matrix}$   
 ✓  $\text{int } [ ] a[], b; \quad \begin{matrix} a \rightarrow 2 \\ b \rightarrow 1 \end{matrix}$   
 ✓  $\text{int } [ ] a[], b[]; \quad \begin{matrix} a \rightarrow 2 \\ b \rightarrow 1 \end{matrix}$   
 ✓  $\text{int } [ ] a[], b[]; \quad \begin{matrix} a \rightarrow 2 \\ b \rightarrow 2 \end{matrix}$   
 ✓  $\text{int } [ ] a[], b[]; \quad \begin{matrix} a \rightarrow 2 \\ b \rightarrow 3 \end{matrix}$   
 ✗  $\text{int } [ ] a, b; \quad \begin{matrix} a \rightarrow 1 \\ b \rightarrow 2 \end{matrix}$

## Conclusion

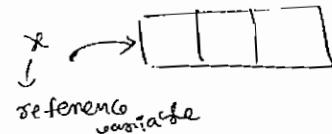
If we want to specify dimensions before the variable that facilities available only for the first variable. If we trying to apply for the remaining variable then we will get compiletime error.

✗  $\text{int } [ ] a, b, c;$   
 ✓  $\text{int } [ ] a, [ ] b, [ ] c;$   
 ✗  $\text{int } [ ] a, [ ] b, [ ] c;$   
 ✗  $\text{int } [ ] a, [ ] b, [ ] c;$

### ③ Array Creation

11-08-2014 Every Array in Java is an object and hence we can create by using new operator.

`int[] x = new int[3];`



→ for every array type corresponding classes are available but these classes are part of Java language and not available to the programmer level.

Eg: `int[] x = new int[3];  
System.out.println(x.getClass().getName()); // [I`

Arraytype	Corresponding class name
<code>int[]</code>	<code>[I</code>
<code>byte[]</code>	<code>[C</code>
<code>double[]</code>	<code>[D</code>
<code>byte[]</code>	<code>[B</code>
<code>short[]</code>	<code>[S</code>
<code>boolean[]</code>	<code>[Z</code>

1) At the time of Array Creation compulsory we have to specify the size otherwise we will get Compiletime error

Eg: `int[] x = new int[]; X C.O.G  
int[] x = new int[3]; ✓`

2) It is illegal to have an array with size 0 (zero)

Eg: `int[] x = new int[0]; ✓`

class: `(class_name.java)`

file: `(file_name.java)`

source code: `(source_code.java)`

Java file: `(java_file.java)`

Java code: `(java_code.java)`

file: `(file.java)`

Java file: `(java.java)`

Java code: `(java_code.java)`

file: `(file.java)`

Java file: `(java.java)`

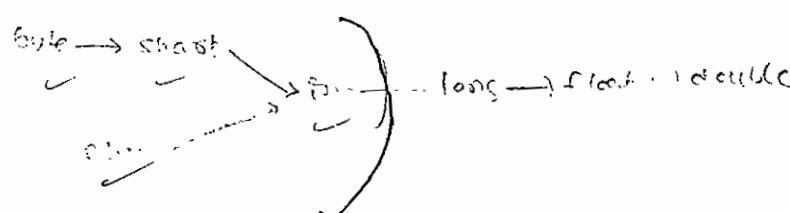
Java code: `(java_code.java)`

3) if we are trying to specify Arraysize with negative int value then we will get Runtime exception saying Negative Array Size Exception

Eg: `int[] x = new int[-3];  
Compiles X R.E: Negative ArraySizeException`

4) To specify Arraysize the allowed datatypes in Java are

`byte  
short  
char  
int`



← what data are allocated

- If we are using any datatype then we will get Compiletime error.

`int[] x = new int[10]; ✓`

`int[] x = new int['a']; ✓`

`byte b = 20; ✓`

`int[] x = new int[6]; ✓`

`short s = 30; ✓`

`int[] x = new int[8]; ✓`

`int[] x = new int[100]; X`

CSE

Possible loss of precision

found: long  
required: int

(5)  
WORK

The maximum allowed Arraysize in java is 2147483647, which is the maximum value of int datatype.

Eg 1) `int[] x = new int[2147483647]; ✓`

2) `int[] x = new int[2147483648]; X`

CSE

{ Integer number too large

→ even in the first case we may get OutofmemoryError if sufficient ~~real~~ memory not available in our system

(2147483647+1, 2147483648)

### Multidimensional Array creation

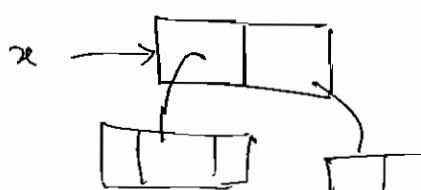
- In Java multidimensional arrays are not implemented by using Matrix representation.
- SUN people followed Array of arrays approach to design multidimensional arrays.
- The main advantage of this approach of memory utilization will be improved.

Eg: 1

`int[][] x = new int[2][2];`

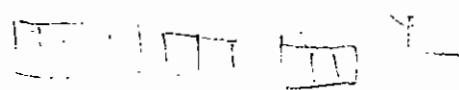
`x[0] = new int[2];`

`x[1] = new int[2];`



s <sub>1</sub>	10	17	18	19	20
s <sub>2</sub>	60	48	X	X	X
s <sub>3</sub>	70	45	32		
s <sub>4</sub>	70	X	Y		

s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	s <sub>4</sub>
----------------	----------------	----------------	----------------

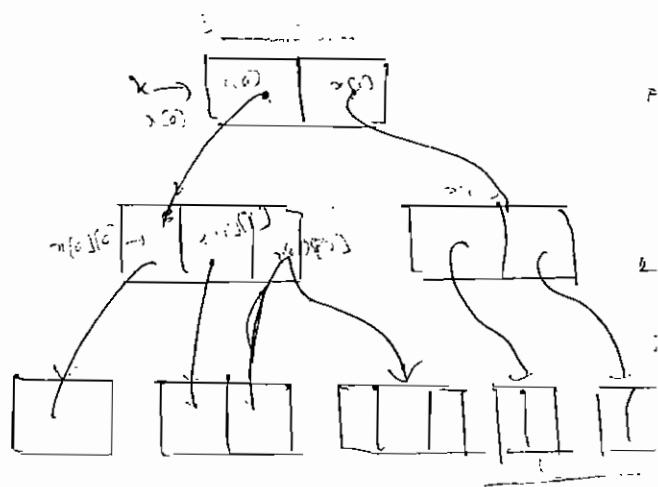


Eg 2.  $\text{int } a = \text{new int}[3][3];$

```

 $a[0] = \text{new int}[2][2];$ 
 $a[0][0] = \text{new int}[1][1];$ 
 $a[0][1] = \text{new int}[2][2];$ 
 $a[0][2] = \text{new int}[3][3];$ 
 $a[1] = \text{new int}[2][2]$ 

```



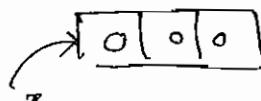
Which of the following are valid

- ①  $\text{int } a = \text{new int}[3];$  X
  - ②  $\text{int } a = \text{new int}[2];$  ✓
  - ③  $\text{int } a = \text{new int}[5][5];$  X
  - ④  $\text{int } a = \text{new int}[3][3];$  ✓
  - ⑤  $\text{int } a = \text{new int}[3][4];$  X
  - ⑥  $\text{int } a = \text{new int}[3][4];$  ✓
  - ⑦  $\text{int } a = \text{new int}[3][4][5];$  ✓
  - ⑧  $\text{int } a = \text{new int}[3][4][5];$  ✓
  - ⑨  $\text{int } a = \text{new int}[3][4][5];$  ✓
  - ⑩  $\text{int } a = \text{new int}[3][3][5];$  X
  - ⑪  $\text{int } a = \text{new int}[3][4][5];$  X
- Notes: 1. no second large size without reason  
2. can't declare 3rd dimension

#### ④ Array Initialization

- ① Once we creates an array every array element is by default initialized with default values

Eg 1.  $\text{int } a = \text{new int}[3];$   
 $\text{SOP}(a);$  [I@32255  
 $\text{SOP}(a[0]);$  A 0



#### Notes:

When ever we are trying to print any object reference internally `toString` method will be called which is by default implemented to return string like the following form

Class name @ hashCode - in hexadecimal form

12-08-14

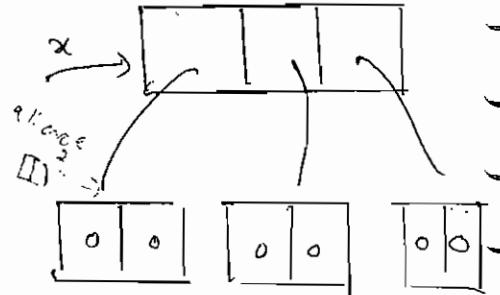
Ex 28

`int x[ ] x = new int[3][2]`

`Sop(x); O/P → [ [ I @3e25a5`

`Sop(x[0]); [ I @1821f`

`Sop(x[0][0]); 0 ↗ one dimensional array`



[ C @3e25a5 ]

int x[] ↗ two dimensions of array

Ex 29

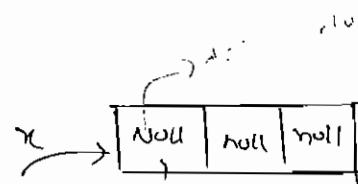
`int x[ ][ ] x = new int[3][3];`

`Sop(x); [ C I @12b6651`

`Sop(x[0]); null`

`Sop(x[0][0]); R.E.`

null pointer exception



Note 8

If we are performing any operation on null

then we will get NullPointerexception

→ Once we creates an array every array element by default initialized with default values. If we are not satisfied with default values then we can override default values with our customized values

Eg:

`int x = new int[5];`

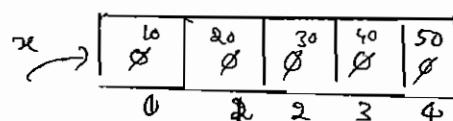
`x[0] = 10;`

`x[1] = 20;`

`x[2] = 30;`

`x[3] = 40;`

`x[4] = 50;`



`x[5] = 60; R.E.: ArrayIndexOutOfBoundsException`

`x[-5] = 70; R.E.: ArrayIndexOutOfBoundsException`

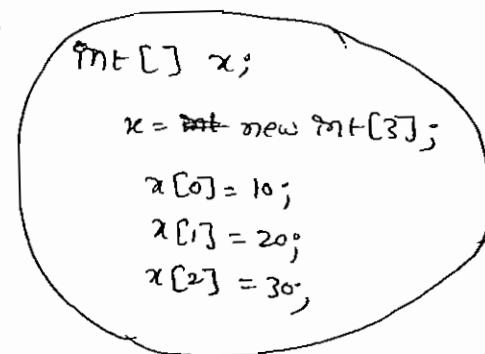
`x[2.5] = 80; C.E.:`

Possible loss of precision  
found: double  
required: int

⑤ Array declaration, creation and initialization in a single line

- we can declare, create and initialize an array in a single line (shortcut representation)

Ex:



```
int[] x = {10, 20, 30};
```

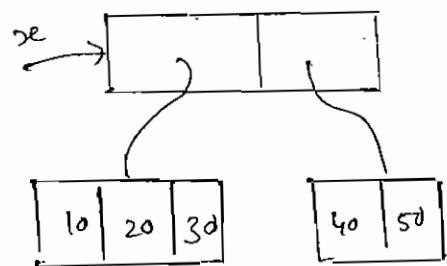
```
char[] ch = {'a', 'e', 'i', 'o', 'u'};
```

```
String[] s = {"A", "AA", "AAA"};
```

→ We can use this shortcut even for multidimensional arrays also.

Ex:

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

but

```
int[][][] x = {{{10, 20, 30}, {40, 50}}, {{60, 70, 80}, {90, 100, 110}}};
```

```
Sop(x[0][0][0]); 90
```

```
Sop(x[0][2][0]); R.E: ArrayIndexOutOfBoundsException
```

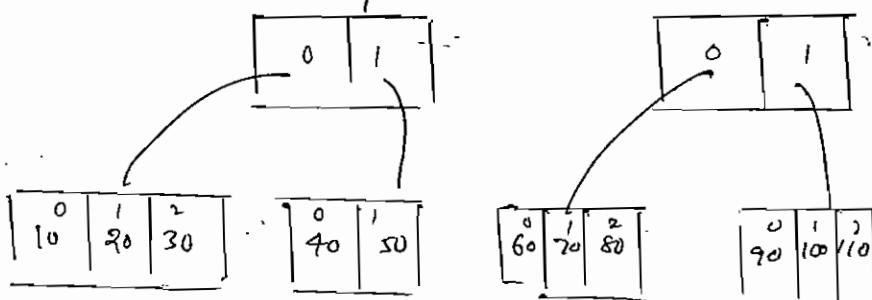
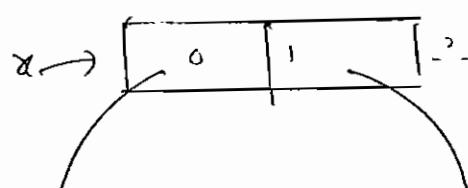
```
Sop(x[2][1][0]); R.E: ArrayIndexOutOfBoundsException
```

```
Sop(x[0][1][2]); R.E: ArrayIndexOutOfBoundsException
```

```
Sop(x[1][0][2]); 80
```

```
Sop(x[2][0][1]); R.E: ArrayIndexOutOfBoundsException
```

```
Sop(x[1][0][1]); 100
```



~~\*\*~~ If we want to use this Shortcut Compulsory we should Perform all activities in a single line. If we are trying to divide into multiple lines then we will get compilation error.

Ex:

```
int[] x = {10, 20, 30};  
          ^
```

```
int[] x;
```

```
x = {10, 20, 30};
```

~~GE~~

Illegal start or expression

114 x=10;

115

int x;

116 x=10

(6)

length vs length()

- length is a final variable applicable for arrays
- length variable represents the size of the array

Ex:

```
int[] x = new int[8];
```

```
Sop(x.length()); C:\E:\
```

```
Sop(x.length()); /13
```

Can not find symbol  
Symbol: method length()  
Location: class int[]

length():

- length() method is a final method applicable for String objects
- length() method returns number of characters present in the String

Ex:

```
String s = "durga";
```

```
Sop(s.length()); C:\E:\
```

```
Sop(s.length()); /15
```

Can not find symbol  
Symbol: variable length  
Location: class java.lang.String

Note: length variable applicable for Arrays but not for String objects whereas length() applicable for String objects but not for Arrays.

$s[0]$  is a string

## Q1

String s = {"A", "AA", "AAA"};

✓ `Sop(s.length); // 3`

✗ `Sop(s[0].length);`

✗ `Sop(s.length());`

✓ `Sop(s[0].length()); // 1`

CCE → Can not find symbol  
symbol: variable length  
location: class java.lang.String

CCE → Can not find symbol  
symbol: method length()  
location: class String

## Q2

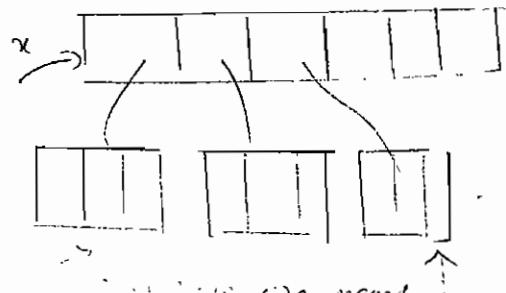
### Notes

In multidimensional arrays length variable represents only base size but not total size

#### Ex:

`int[][] x = new int[6][3];`

`Sop(x.length); // 6`



↑ base size means

In multidimensional array

The following base size will be  
not fixed.

There is no direct way to find total size of 2-dimensional/multidimensional array but indirectly we can find as follows.

$x[0].length + x[1].length + x[2].length + \dots$

(7)

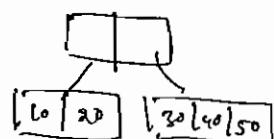
## Anonymous Arrays

- Some times we can declare an array without name such type of nameless arrays are called anonymous arrays
- The main purpose of anonymous arrays is just for instant use `[ ]`
- We can create anonymous array as follows

`[new int[] {10, 20, 30, 40}]`

- We can create multidimensional anonymous arrays also

`[new int[][] {{10, 20}, {30, 40, 50}}]`



while creating anonymous arrays we can't specify the size otherwise we will get compilation error.

- \* new int[3] {10, 20, 30}
- ✓ new int[] {10, 20, 30}

based on our requirement we can give the name for anonymous arrays then it is no longer anonymous.

Ex:

```
int[] x = new int[] {10, 20, 30};
```

Ex8

```
class Test
{
    public static void main(String[] args)
    {
        sum(new int[] {10, 20, 30, 40});
    }

    public static int sum(int[] x)
    {
        int total = 0;
        for (int i = 0; i < x.length; i++)
        {
            total = total + x[i];
        }
        System.out.println("The sum is " + total);
    }
}
```

In the above example just to call sum() method we required an array but after completing sum() method call we are not using that array any more for this instant use (one time usage) anonymous array is best suitable.

### (8) Array element assignments

Case 1

In the case of primitive type arrays as array elements we can provide any type which can be implicitly promoted to declared type.

Ex1

for int type arrays allowed element types are byte, short, char, int

Ex2

for float type arrays allowed element types are byte, short, char, int, long, float.

Ex:

```

int[] x = new int[10];
x[0] = 70;
x[1] = 'a';
byte b = 20;
x[2] = b;
short s = 30;
x[3] = s;
X x[4] = 10.0; → C-E

```

byte → short

char

int → long → float → double

Possible Loss of Precision

→ found: long  
required: float

Case 2:

- In the case of object type arrays as array elements we can provide either declared type objects (or) it's child class objects

Ex:

```

Object[] a = new Object[10];
a[0] = new Object();
a[1] = new String("durga");
a[2] = new Integer(10);

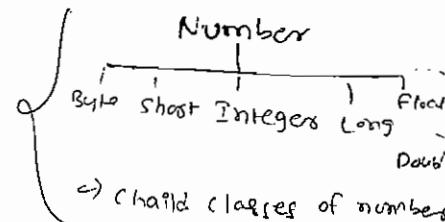
```

Number[] n = new Number[10];

n[0] = new Integer(10);

n[1] = new Double(10.5);

X n[2] = new String("durga");



Abstract class

Incompatible types

found: java.lang.String

required: java.lang.Number

{ " Runnable " interface we can not create object

Case 3:

- In the case of interface type arrays as array elements we can provide it's implemented class objects

{ " Runnable " interface we can not create object

Runnable[] r = new Runnable[10];

r[0] = new Thread();

r[1] = new Integer(10); X

Runnable(II)

Thread

C-E: Incompatible types

found: java.lang.Integer

required: java.lang.Runnable

## Array type

### Allowed element type

① Primitive type

Array type which can be implicitly provided to declared type

② Object type

either declared type or its child class objects

③ Abstract class type

its child class objects

④ Interface type

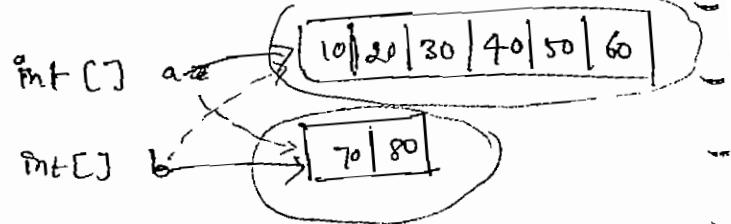
its implemented class objects are allowed.

## Q) Array variable Assignments

Case 1 whenever we are assigning one array to another array internal elements won't be copied just reference variables will be reassigned hence sizes are not required to match but types must be matched.

Ex:  
int [] a = { 10, 20, 30, 40, 50, 60 } ;  
int [] b = { 70, 80 } ;

a = b ;  
b = a ;



(21-08-14)

Case 2

Array element level promoting are not applicable at Array Object Level. for example char element can be promoted to int type but char[] array can not be promoted to int[] (int array) type.

Ex: int [] a = { 10, 20, 30, 40 } ;  
char [] ch = { 'a', 'b', 'c', 'd' } ;

int [] b = a ; ✓

int [] c = ch ; X   
C & ch are of incompatible types  
found : char []  
required : int []

char [] a ;  
char [] b ;  
char [] c ;  
char [] d ;

[c]

Which of the following conversions will be performed automatically (31)

char → int

X char[] → int[]

✓ int → float

X int[] → float[]

X double → int

X double[] → int[]

✓ String → Object

✓ String[] → Object[]

Child to parent

Object from a child class

But in the case of Object type Arrays Child class type array can be assigned to its Parent type array reference variable.

Eg:

String[] s = {"A", "B", "C"};

✓ Object[] a = s;

### Cafe 3.5

Whenever we are assigning one array to another array dimensions must be matched.

for example in the place of one dimensional array we should provide one dimensional array only, if we are trying to provide any other dimension then we will get compile time error.

✓ int[][] a = new int[3][];

X a[0] = new int[3][2];

X a[0] = 10;

a[0] = new int[2];

Incompatible types

found: int[3][2]

required: int[]

[[1, 1, 1]]

c.c

Incompatible types

found: int

required: int[]

[1, 1]

### Note:

Whenever we are assigning one array to another array dimensions must be matched. but sizes of arrays and types are not required to match.

Ex 28

```

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

```

```

for (int i=0, i<=args.length; i++)
{
    System.out.println(args[i]);
}

```

Java Test A B C ↴

R.E → ArrayIndexOutOfBoundsException  
Test A B ↴

R.E → ArrayIndexOutOfBoundsException

Java Test ↴

R.E → ArrayIndexOutOfBoundsException

If we replace  $\leq$  with  $<$  then we won't get any runtime exception

Ex 28

```

class Test
{
    public static void main(String[] args)
    {
        String[] args = {"x", "y", "z"};
        System.out.println(args);
        for (String s : args)
        {
            System.out.println(s);
        }
    }
}

```

args ("x|y|z")

args [x|y|z]

args, [x|y|z]  
args [x|y|z]

Java Test A B C ↴

Java Test A B ↴

Java Test ↴

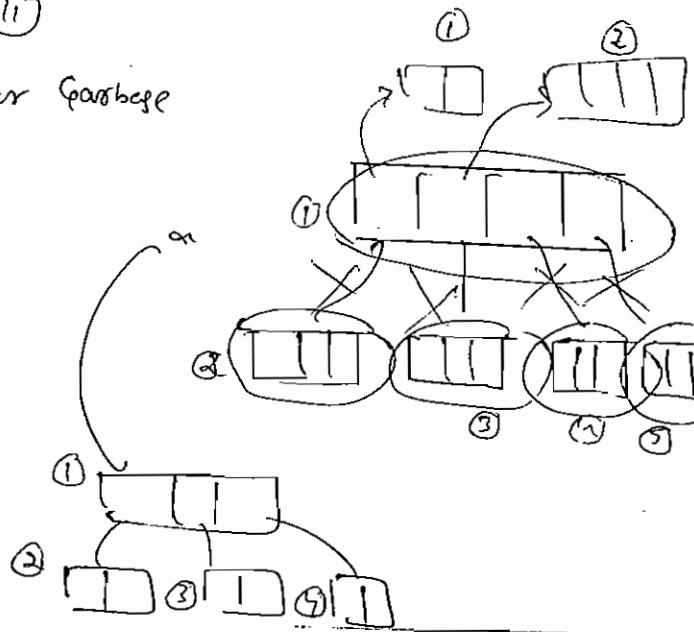
Ex 29

```

int[][] a = new int[4][3]; → ⑤
a[0] = new int[2]; → ①
a[1] = new int[4]; → ②
a = new int[3][2]; → ③

```

- ① Total how many objects created? (11)  
 ② How many objects are eligible for Garbage Collection? (7)



## ⑥ Types of Variables

### Division-I

Based on type of data value represented by a variable all variables are divided into ② types

#### ① Primitive variables

Can be used to hold primitive values

Ex: int x = 10;

#### ② Reference variables

Can be used to refer objects

Ex: Student s = new Student();

### Division-II



Based on purpose and position of declaration all variables are divided into ③ types

#### ① Instance variables

#### ② Static variables

#### ③ Local variables

#### ① Instance variables:

If the value of a variable is varied from object to object such type of variables are called instance variables

- ② for every object a separate copy of instance variable will be created
- ③ Instance variable will be created at the time of Object creation and instance variable will be destroyed at the time of Object destruction. hence the scope of instance variable is exactly same as the scope of object.
- ④ instance variables will be stored (or) saved in the heap area as the part of object.
- ⑤ instance variables should be declared with in the class directly but outside of any method (or) block (or) constructor
- ⑥ we can't access instance variables directly from static area but we can access by using object reference
- ⑦ we can access instance variables directly from instance area
- Eg:
- ```

class Test {
    int x=10;
    public void main(String[] args) {
        // System.out.println(x); C.E: Non static variable x can not be
        // referenced from a static context
        Test t=new Test();
        System.out.println(t.x);
    }
}

```

- ⑧ for instance variable and we are not required to perform initialization explicitly Jvm will always provide default values
- Eg:
- ```

class Test {
    double d;
    String s;
    boolean b;
    int e;
    public void main(String[] args) {
        Test t=new Test();
        System.out.println(t.d); // 0.0
        System.out.println(t.s); // null
        System.out.println(t.b); // false
        System.out.println(t.e); // 0
    }
}

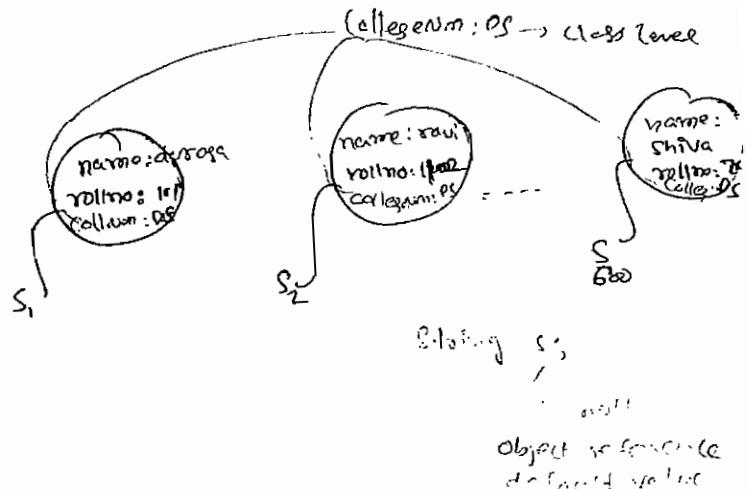
```

① Instance variables also known as Object level variables (or) attribute.

```

class Student
{
    String name;
    int rollno;
    ...
    void m1() // method
    {
        student(); // constructor
    }
    static C1; // field
}

```



### [22-08-14] (2) Static variables

- If the value of a variable is not varied from object to object then it is not recommended to declare that variable as instance variable. We have to declare such type of variables at class level with static modifier.
- In the case of instance variable for every object a separate copy will be created but in the case of static variable a single copy will be created at class level and shared by every object of that class.
- Static variables should be declared with in the class directly but outside of any method or block or constructor.
- Static variables will be created at the time of class loading and destroyed at the time of class unloading hence the scope of static variable is exactly same as scope of class file.

Ex:

#### Java Test

- Start JVM
- Create & Start main thread
- Locate Test.class
- Load Test.class
- Execute main method
- Unload Test.class
- Terminate main thread
- Shutdown JVM

Static variable creation

Static variable destruction

5. Static variables will be stored in method area.

→ We can access static variables either by using object reference or by using class name but recommended to use class name.  
Within the same class even class name not required we can access directly

Ex:

```
class Test
{
    static int x = 10;

    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.x); // 10
        System.out.println(Test.x); // 10
        System.out.println(x); // 10
    }
}
```

\*→ We can access static ~~variables~~ variables directly from both instance and static area.

Ex:

```
class Test
{
    static int x = 10; // Created at the time of class loading
    public static void main(String[] args)
    {
        System.out.println(x); // 10
        public void m1()
        {
            System.out.println(x); // 10
        }
    }
}
```

\*→ for static variables we are not required to perform initialization explicitly JVM will always provide default values

Ex:

```
class Test
{
    static int x;
    static String s;
    static double d;
    public static void main(String[] args)
    {
        System.out.println(x); // 0
        System.out.println(s); // null
        System.out.println(d); // 0.0
    }
}
```

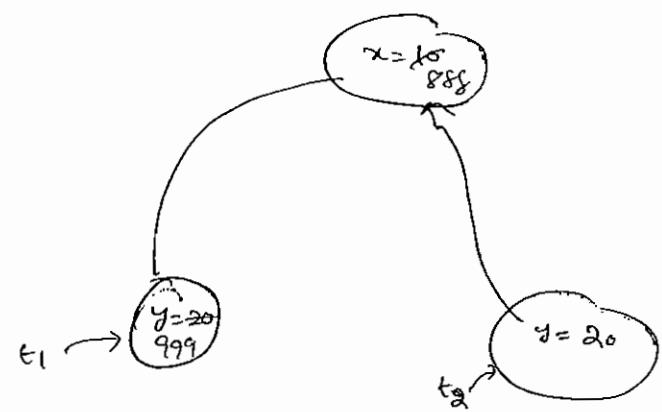
\* Static variables also known as class level variables (or) fields

### Ex 8 Class Test

```

Static int x=10;
int y=20;

P s v main (String[] args)
{
    Test t1 = new Test();
    t1.x=888;
    t1.y=999;
    Test t2 = new Test();
    System.out.println(t2.x+"---"+t2.y);
}
    
```



### (3) Local Variables

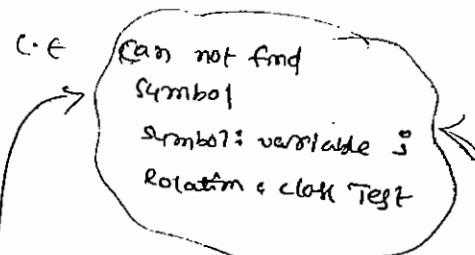
- Some times to meet temporary requirements of the programmer we can declare variables inside a method or block or constructor. Such type of variables are called local variables or stack variables or temporary variables or automatic variables.
- Local variables will be stored inside stack memory.
- Local variables will be created while executing the block in which we declared it. Once block execution completes automatically local variable will be destroyed. Hence the scope of local variable is the block in which we declared it.

### Ex 8

#### Class Test

```

P s v main (String[] args)
{
    int i=0;
    for(int j=0; j<3; j++)
    {
        i=i+j;
    }
    System.out.println(i+"---"+j);
}
    
```



#### Class Test

```

P s v main (String[] args)
{
    try
    {
        int g = Integer.parseInt("4en");
    }
    catch (NumberFormatException e)
    {
        i=10;
    }
    System.out.println(g);
}
    
```

- for local variables JVM won't provide default values compulsory we have to perform initialization explicitly before using that variable
- If we are not using a variable then it is not required to perform initialization explicitly.

Ex:

Class Test

```

l 1 s v main (String[] args)
  l int x;
    y System.out.println("Hello");
  ]
  o/p: Hello
  
```

Class Test

```

l 1 s v main (String[] args)
  l int x;
    y System.out.println(x);
  ]
  
```

C.B Variable x might not have been initialized

Class Test

```

l 1 s v main (String[] args)
  l int x; //int x=0 (note x)
    if (args.length > 0)
      l x=10; (note 1)
    y System.out.println(x);
  ]
  
```

C.B Variable x might not have been initialized.

Class Test

```

l 1 public static void main (String[] args)
  l int x;
    if (args.length > 0)
      l x=10;
    y else
      l x=20;
    y System.out.println(x);
  ]
  o/p Java Test A B
  
```

Java Test ↗

20

Note ①

It is not recommended to perform initialization for local variables inside logical blocks because there is no guarantee for the execution of these blocks at

Note ②

It is highly recommended to perform initialization for local variables at the time of declaration atleast with default values.

Ex: int x=0;  
Console.out.println(x);

(25-08-14)

Note 3: The only applicable modifier for local variable is final. By mistake if we are trying to apply any other modifier then we will get compile time error.

### Class Test

```

    {
        P s v main(String [] args)
        {
            public int x=10;
            private int x=10;
            protected int x=10;
            static int x=10;
            transient int x=10;
            volatile int x=10;
            final int x=10;
        }
    }
  
```

### Note 4:

If we are not declaring any access modifier then by default it is default but this rule is applicable only for instance & static variables but not for local variables.

### Summary

- For instance and static variables JVM will always provide default values and we are not required to perform initialization explicitly. but for local variables JVM won't provide default values compulsorily we have to perform initialization explicitly before using that variable.

### Uninitialized Arrays

#### Class Test

```

    int [] a;
    public static void main(String [] args)
    {
        Test t=new Test();
        System.out.println(t.a); // null
        System.out.println(t.a[0]); // R.E. NullPointerException
    }
  
```

### Summary (Instance Level)

(i) `int [] a;`

`System.out.println(obj.a);`; null

`System.out.println(obj.a[0]);`; R.E.: NPE

② int[] a = new int[3];

SOP(obj.a); [I@3e25a5

SOP(obj.a[0]); 0



### Static Level

① static int[] a;

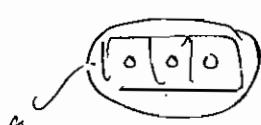
SOP(a); null

SOP(a[0]); R.E: NPE

② static int[] a = new int[3];

SOP(a); [I@3e25a5

SOP(a[0]); 0



### Local Level

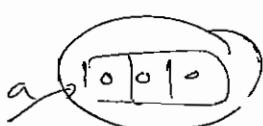
① int[] a;

SOP(a); } CES: Variable 'a'  
SOP(a[0]); } might not have been  
initialized

② int[] a = new int[3];

SOP(a); [I@3e25a5

SOP(a[0]); 0

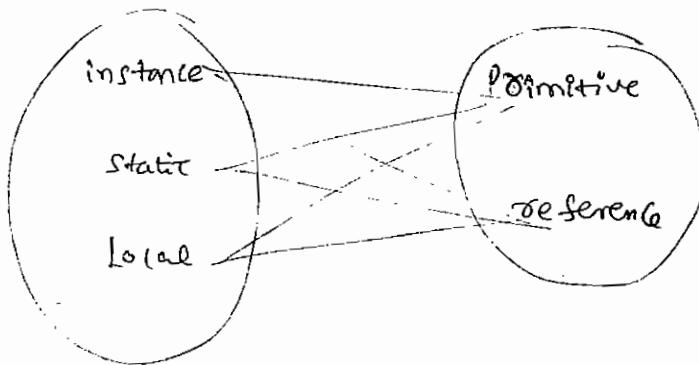


### Notes

once we creates an array every array element by default initialized with default values irrespective of whether it is instance or static or local array.

### Notes

Every variable in Java should be either instance (or) static (or) local hence the following case various possible combinations of variables in Java.

Ex:

```

class Test
{
    instance-Primitive → int x=10;
    static-reference → static String s="durga";
    ↘ s ← main(strings[] args)
    Local-reference → int[] a = new int[i];
    ↘
}
  
```

Instance variables  $\Rightarrow$  Heap Area  
 static variables  $\Rightarrow$  method Area  
 Local variables  $\Rightarrow$  Stack Area

⑦

## Var-arg methods (Variable number of argument methods)

- Until 1.4 version we can't declare a method with variable numbers of arguments. If there is a change in numbers of arguments completely we should go for new method it increases length of the code and reduces readability. To overcome the problem Sun people introduced var-arg methods in 1.5 version according to this we can declare a method which can take variable number of arguments such type of methods are called var-arg methods.
- We can declare a var-arg method as follows

`m1(int... x)`

We can call this method by passing any number of int values including zero numbers.

`m1();` ✓  
`m1(10);` ✓  
`m1(10, 20);` ✓  
`m1(10, 20, 30, 40);` ✓

26-08-14 Class Test

```

P S V main (String[] args)
{
    P S V m (int... x)
    {
        for ("var-arg method");
    }
    P S V main (String[] args)
    {
        m ( )
        m (10)
        m (10, 20)
        m (10, 20, 30, 40);
    }
}

```

Op: 10 20 30

Op: var-arg method  
var-arg method  
var-arg method  
var-arg method

\* Internally var-arg parameter will be converted into one dimensional array hence with in the var-arg method we can differentiate values by using index.

Ex:

Class Test

```

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

```

```

        sum ( );
        sum (10, 20);
        sum (10, 20, 30);
        sum (10, 20, 30, 40);
}

```

Op: The sum is  
The sum is 30  
The sum is 60  
The sum is 100

public static void sum (int ... x)

```

{
    int total = 0;
    for (int x1 : x)
    {
        total = total + x1;
    }
}

```

Op ("The sum is" + total);

Ques 1

which of the following valid var-arg method declarations

m (int [] x)  
m (int [ ] x)  
m (int x [])

m (int ... x) ✓
m (int ... x) ✓
m (int ... x) ✓
m (int x ...) ✗
m (int ... x) ✗
m (int ... x) ✗

Case 2: we can mix var-arg Parameter with normal Parameter

$$\left\{ \begin{array}{l} m_1(\text{int } x, \text{ int... } y) \\ m_1(\text{String } s, \text{ double... } y) \end{array} \right.$$

Case 3: if we mix normal parameter with var-arg Parameter then var-arg Parameter should be last parameter.

$m_1(\text{double... } d, \text{ String } s)$  X

$m_1(\text{char } ch, \text{ String... } s)$  ✓

Case 4: Inside var-arg method we can take only one var-arg parameter and we can't take more than one var-arg parameter

$m_1(\text{int... } x, \text{ double... } d)$  X

Case 5: Inside a class we can't declare one dimensional array method var-arg method and corresponding simultaneously otherwise we will get compilation error.

Class Test  
P = v  $m_1(\text{int... } x)$   
y  
P = v  $m_1(\text{int[ ] } x)$   
y

C.o.E  
Can not declare both  $m_1(\text{int[ ] } x)$  and  $m_1(\text{int... } x)$  in Test

Case 6: Class Test

P = v  ~~$m_1(\text{int... } x)$~~   
SOP ("var-arg method");  
P = v  $m_1(\text{int } x)$   
SOP ("General method");  
P = v  $\text{main(String[ ] args)}$   
 $m_1();$  var-arg method  
 $m_1(10, 20);$  var-arg method  
 $m_1(10);$  General method

In general var-arg method will set least priority i.e if no other method matched then only var-arg method will get the chance. It is exactly same as default case inside switch.

### Equivalence b/w var-arg Parameter and one dimensional array

Case1: wherever one dimensional array present we can replace with var-arg parameter

$$m_1(\text{int}[] \underline{x}) \Rightarrow m_1(\text{int... } \underline{x})$$

Ex1

`main(String[] args)` can be replaced with `main(String... args)`

Case2:

wherever array var-arg parameter present we can't replace with one dimensional

$$\times \boxed{m_1(\text{int... } \underline{x}) \Rightarrow m_1(\text{int}[] \underline{x})}$$

$$\begin{array}{|c|c|} \hline m_1(\text{float... } \underline{x}) & m_1(\text{int}[] \underline{x}) \\ \hline m_1(); & \times \\ \hline m_1(10); & \times \\ \hline m_1(10, 20); & \times \\ \hline m_1(10, 20, 30); & \times \\ \hline \end{array}$$

### Notes

① `m_1(int ... x)`

We can call this method by passing a group of int values and x will become one dimensional array `(int[] x)`

$$\begin{array}{|c|} \hline m_1(\text{int}[] \underline{x}) \\ \hline 10, 20, 30 \\ \hline \end{array}$$

② `m_1(int[] ... x)`

We can call this method by passing a group of one dimensional int arrays and x will become a dimensional int array `(int[][] x)`

Ex2 class Test

{  
    P sv main(String[] args)

    d  
        int[] a = {10, 20, 30};

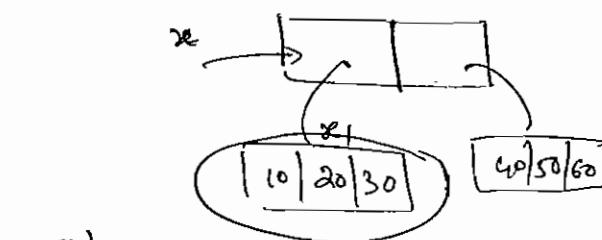
        int[] b = {40, 50, 60};

        m(a, b);

    y  
        public static void m(int[] ... x)

    f  
        for (int[] z, e z)

        d  
            sop(z[0]);



    y  
        y  
            sop(z[0]);

        o/p [ 10 ]

⑧

Main Method

whether class contains main() method or not and whether main() method is declared according to requirement or not these things can't be checked by Compiler at runtime JVM is responsible to check these things if JVM unable to find main() method then we will get **RuntimeException** saying **NoSuchMethodError: main**

Ex:

Class Test

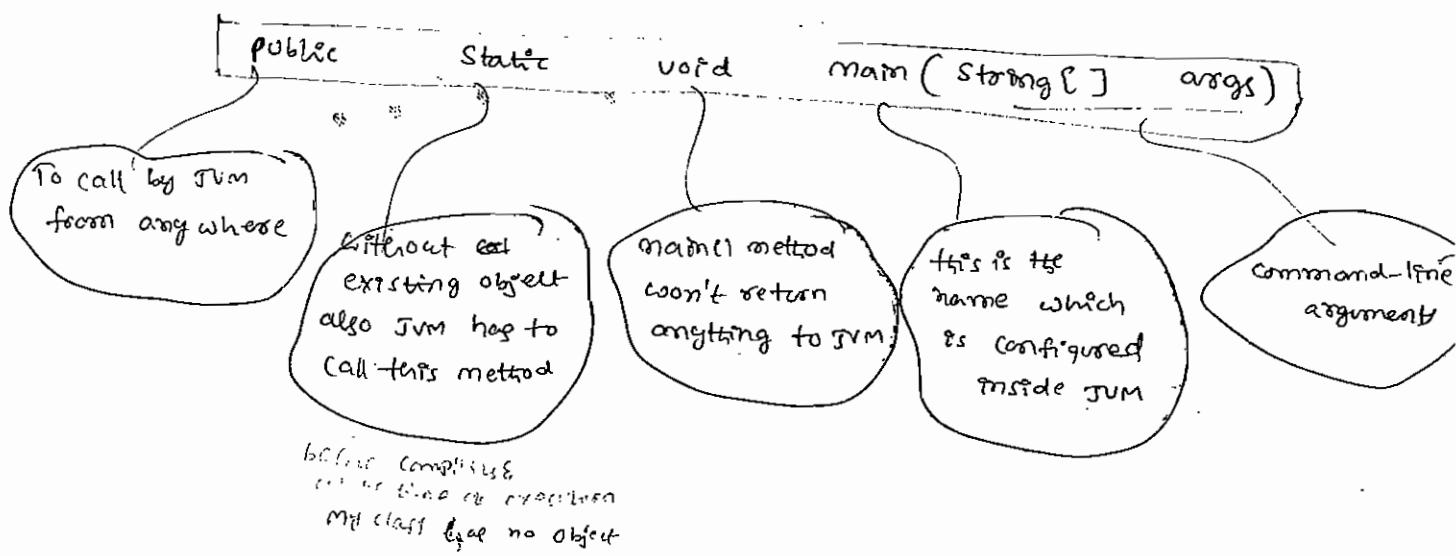
3

javac Test.java ✓

Java Test ↴

R.E.: NoSuchMethodError: main

→ At runtime JVM always searches for the main() method with the following prototype



→ The above syntax is very strict and if we perform any change then we will get **RuntimeException** saying **NoSuchMethodError: main**

→ Everything above syntax is very strict the following changes are acceptable.

- ① Instead of Public static we can take static Public i.e. the order of modifiers is not important
- ② We can declare `String[]` in any acceptable form

`main(String[] args) ✓``main(String []args) _``main (String args[]) _`

③ Instead of args we can take any ~~for~~ valid java identifiers

Ex: main(String[] durga) ✓

④ we can replace String[] with var-arg Parameter

Ex: main(String... args)

\* we can declare main() method with the following modifiers

final, synchronized  
Strictfp

class Test

```
static final synchronized strictfp public void main(String... durga){  
    System.out.println("Valid main method");  
}
```

① Which of the following main() method declarations are valid

① public static void main (String args) X

② public static void main (String[ ] args) X

③ public void main (String [ ] args) X

④ public static int main (String [ ] args) X

⑤ final synchronized public void main (String [ ] args) X

⑥ final synchronized strictfp public void main (String [ ] args) X

⑦ public static void main (String... args) ✓

② In which of the above cases we will get compiletime error

We won't get compile-time error

In remaining we will get runtime exception saying NoSuchMethodError main

### Case 1

Overloading of the main() method is possible but JVM will always call String[] argument main() method only. The other overloaded method we have to call explicitly like normal method call

Ex:

class Test

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

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

Op: String[]

overloaded methods

Case 2

Inheritance concept applicable for main() method hence ~~of that~~  
while executing child class if child doesn't contain main() method then  
parent class main() method will be executed

Exe

Class P

{

P s v main (String [] args)

{

sop ("Parent main");

{

Class C extends P

{

{

{

javac P.java

P.class

C.class

java P

o/p: Parent main

java C

o/p: Parent main

Case 3

Class P

{

P s v main (String [] args)

{

sop ("Parent main");

{

It's method  
hiding  
but not  
overriding

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

javac P.java

P.class

C.class

java P

o/p: Parent main

java C

o/p: Child main

It seems overriding concept applicable for main() method  
but it is not overriding and it is method hiding

Notes

for main() method inheritance & overloading concepts are applicable  
but overriding concept is not applicable instead of overriding  
the method hiding concept is applicable.

28-08-14

1.7v Enhancements with respect to main() method

javac -verbose  
java -verbose  
java -version

until 1.6v if the class doesn't contain main() method then we will get  
RuntimeException saying NoSuchMethodError: main  
but from 1.7v onwards instead of NoSuchMethodError we will get more  
elaborated error information

Ex 6

Class Test

{

}

1.6v

JavaC Test.java ✓

Java Test ↴

Re: No such method error: main

②

From 1.7 version onwards main() method is mandatory to start program execution hence even though class contains ~~main method~~ static block it won't be executed if the class doesn't contain main() method

Ex 8

Class Test

{

    static {

        System.out.println("Static Block");

}

1.6v

JavaC Test.java ✓

Java Test ↴

Output: Static Block

Re: No such method error: main

1.7v

JavaC Test.java ✓

Java Test ↴

Errors:

Main method not found in class Test,  
Please define the main method as:  
public static void main (String[] args)

1.7v

JavaC Test.java ✓

Java Test ↴

Errors: Main method not found in class Test

③

Ex 26

Class Test

{

    static

{

    System.out.println("Static Block");

    System.exit(0); // Shutdown the JVM

}

}

1.6v

JavaC Test.java ✓

Java Test ↴

Output: Static Block

1.7v

JavaC Test.java ✓

Java Test ↴

Errors: Main method not found in class Test,  
Please define the main method as,  
public static void main (String[] args).

1.0.102 OR 1.0.10  
for checkin on 1.0.10  
1.0.10 or 1.0.10

class Test  
{  
 static  
 {  
 System.out.println("static Block");  
 }  
 public static void main(String[] args)  
 {  
 System.out.println("main method");  
 }  
}

1.6V

Y

Y

1.7V

Javac Test.java ✓

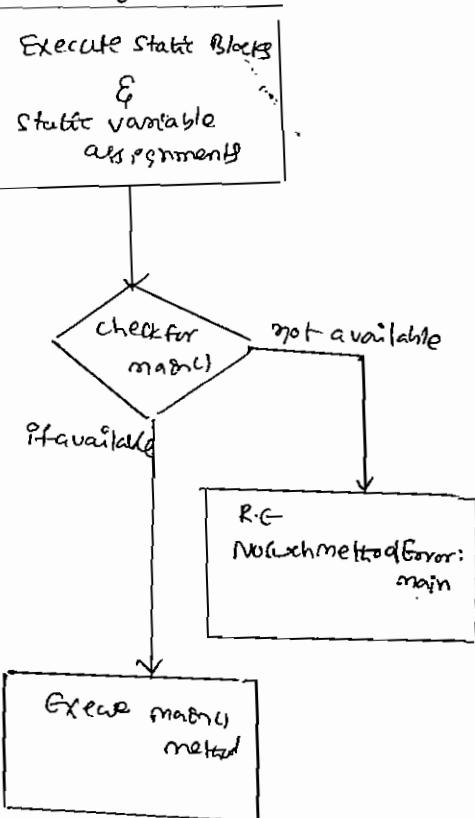
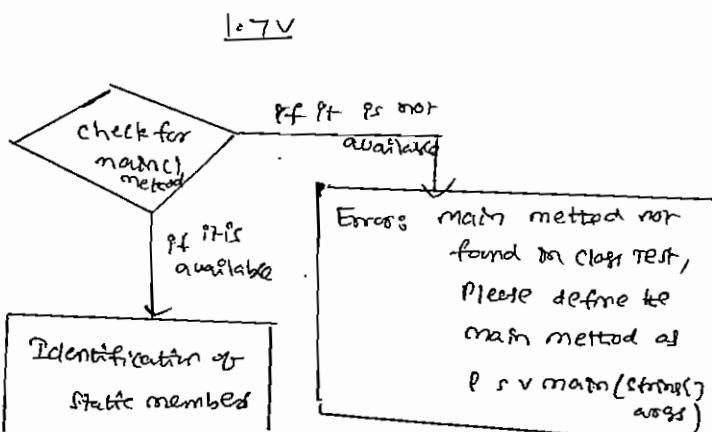
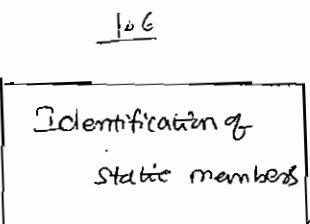
java Test ↴

Output:  
static Block  
main method

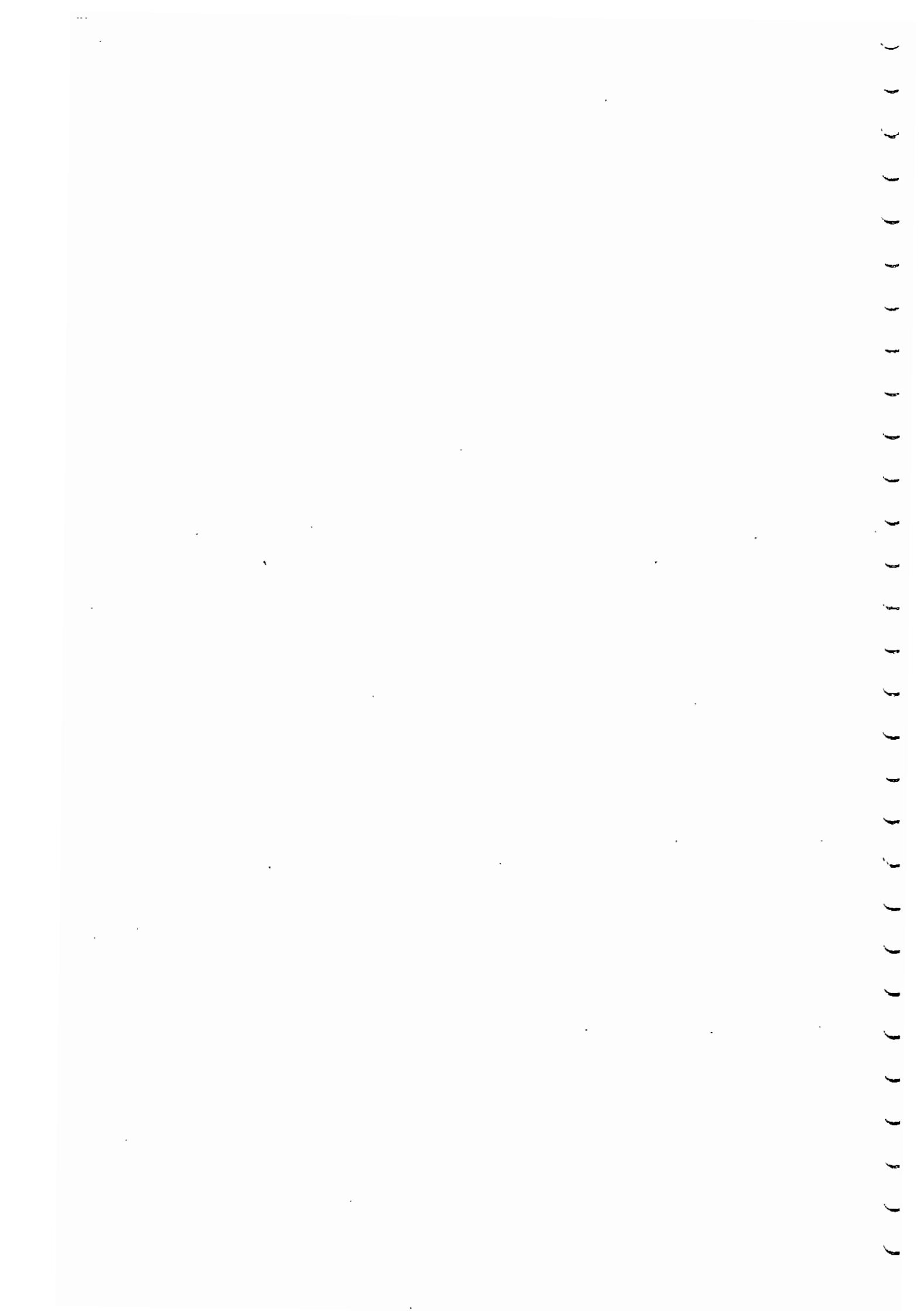
Javac Test.java ✓

java Test ↴

Output:  
static Block  
main method



- (Q) Without writing main() method is it possible to print some statements to the console?  
Yes by using static block but this rule is applicable until 1.6 version but from 1.7 onwards it is impossible to print some statements to the console without writing main() method.

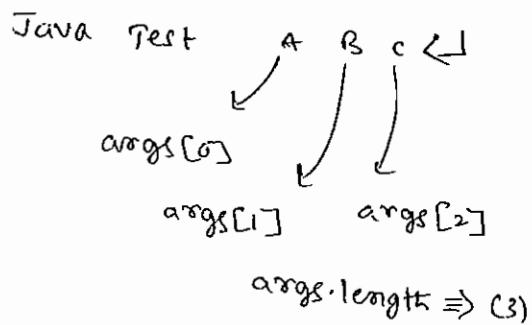


## Command Line Arguments

(3)

- The arguments which are passing from command prompt are called Commandline arguments.
- With these commandline arguments JVM will create an array and by passing that array as argument JVM will call main() method.

Ex:



- The main objective of commandline arguments is we can customize behaviour of the main() method

Case 1

Ex1: class Test

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

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

```
            System.out.println(args[i]);
        }
```

y y y

If we replace  
length or earlier to  $\leq$   
to  $<$  with  $\leq$   
then we won't get any  
runtime exception

Java Test A B C

A  
B  
C

R-E ArrayIndexOutOfBoundsException

Java Test A B

A  
B

R-E ArrayIndexOutOfBoundsException

Java Test

R-E ArrayIndexOutOfBoundsException

Case 2  
class Test

{  
    public static void main(String[] args)

{  
    String[] args = {"x", "y", "z"};  
    args = args;

    } → ①  
    for (String s : args)  
    {  
        System.out.println(s);  
    }

} } }

args → A|B|C

args → X|Y|Z

args → ①

args → X|Y|Z

Java Test A B C ↴

X

Y

Z

Java Test A B ↴

X

Y

Z

Java Test ↴

X

Y

Z

Case 3  
with no arguments in main() method Commandline are available in String form

Eg:  
class Test

{  
    public static void main(String[] args)  
    {  
        System.out.println(args[0] + args[1]);  
    }

Java Test ↴ (o 20)  
OP: o20

Case 4  
usually space & self is the separator b/w commandline arguments  
If our commandline argument itself contains a space then we have to enclose that commandline argument with in double quotes

Eg:  
class Test

{  
    public static void main(String[] args)  
    {  
        System.out.println(args[0]);  
    }

Java Test "note book" ↴

# Java Coding Standards

(23)

Whenever we are writing Java code it is highly recommended to follow coding standards.

Whenever we are writing any component its name should reflect the purpose of that component (functionality).

The major advantage of this approach is readability & maintainability of the code will be improved.

Ex:

Class A

```
public int m1(int x, int y)  
{  
    return x+y;  
}
```

Amespet standard

```
Package com.durgsoft.scp;  
public class Calculator  
{  
    public static int add(int number1,  
                         int number2)  
    {  
        return number1 + number2;  
    }  
}
```

Hi-tech city standard

## Coding Standards for Classes

- Usually class names are nouns
- Should starts with uppercase character and if it contains multiple words every innerword should starts with uppercase character

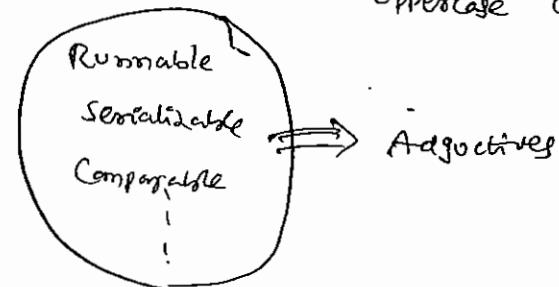
Ex:



## Coding Standards for Interfaces

- Usually interface names are adjectives
- Should starts with uppercase character and if it contains multiple words every innerword starts with uppercase character

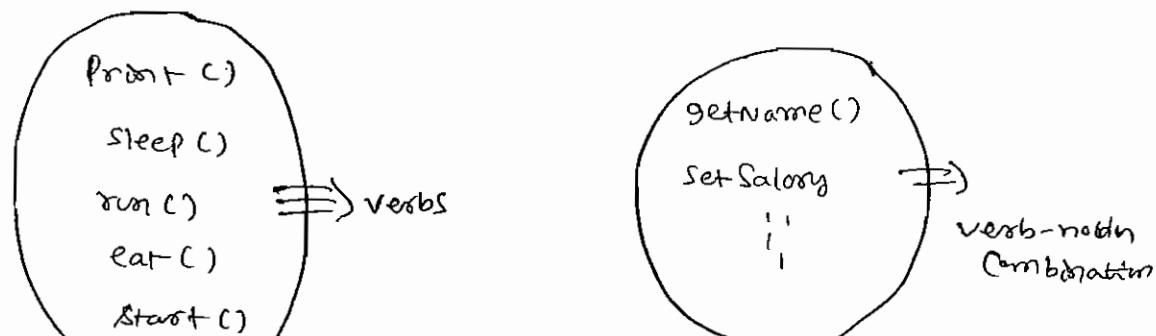
Ex:



## Coding Standards for methods

- Usually method names are either verbs or verb-noun combinations
- should starts with lowercase alphabet symbol and if it contains multiple words then every afterward should starts with uppercase character (CamelCase convention)

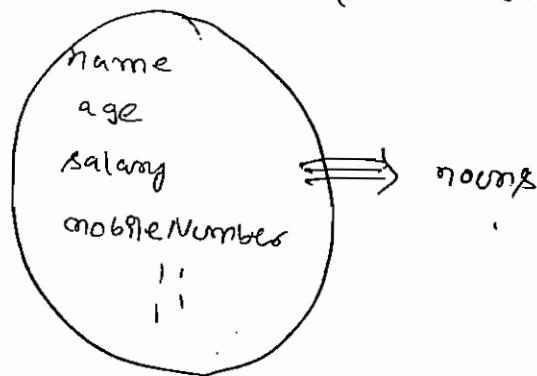
Ex:



## Coding Standards for variables

- Usually variable names are nouns
- should starts with lowercase alphabet symbol and if it contains multiple words then every afterward should starts with uppercase character (CamelCase convention)

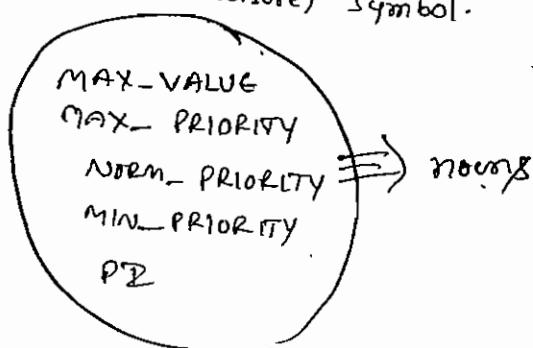
Ex:



## Coding Standards for Constants

Usually constant names are nouns should <sup>contain</sup> starts with only uppercase characters and if it contains multiple words then these words are separated with - (underscore) symbol.

Ex:



Note: Usually we can declare Constants with public static and final modifiers (57)

### Java-Bean Coding Standards:

A JavaBean is a simple Java class with private properties and public getter & setter methods.

Public class StudentBean

```
    {
        Private String name;
        Public void setname(String name)
        {
            this.name = name;
        }
        Public String getname()
        {
            return name;
        }
    }
```

Class name  
ends with 'Bean'  
is not official  
Convention from SUN

#### Syntax for Setter method

- ① It should be public method
- ② The return type should be void
- ③ Method name should be prefixed with set
- ④ It should take some argument i.e. it should not be no argument method.

#### Syntax for Getter method

- ① It should be public method
- ② The return type should not be void
- ③ Method name should prefixed with get
- ④ It should not take any argument

Public void setname(String name)

Public String getname()

#### Note:

for boolean properties Getter method name can be prefixed with either get or is but recommended to use is

Ex:

```

private boolean empty;
public {
    public boolean getEmpty() {
        return empty;
    }
    public boolean isEmpty() {
        return empty;
    }
}

```

both are valid  
recommended

## Coding standards for listeners

Case 1:

**To register a Listener**

Method name should be prefixed with add

Public void	<u>addMyActionListener</u> ( <u>MyActionListener</u> l)
Public void	<u>registerMyActionListener</u> ( <u>MyActionListener</u> l)
Public void	<u>addMyActionListener</u> ( <u>ActionListener</u> l)

Case 2:

**To unregister a Listener**

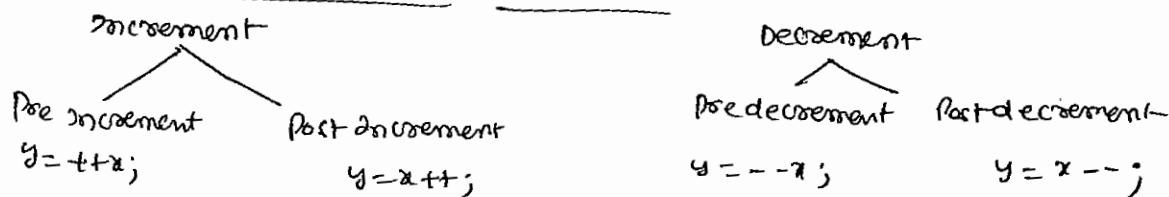
Method name should be prefixed with remove

✓ Public void	<u>removeMyActionListener</u> ( <u>MyActionListener</u> l)
✗ Public void	<u>unRegisterMyActionListener</u> ( <u>MyActionListener</u> l)
✗ Public void	<u>removeMyActionListener</u> ( <u>ActionListener</u> l)
✗ Public void	<u>deleteMyActionListener</u> ( <u>MyActionListener</u> l)

## Operators & Assignments

- ① Increment and Decrement operators
- ② Arithmetic operators
- ③ String Concatenation operators
- ④ Relational operators
- ⑤ Equality operators
- ⑥ instanceof operators
- ⑦ Bitwise operators
- ⑧ Short circuit operators
- ⑨ Type cast operators
- ⑩ Assignment operators
- ⑪ Conditional operators
- ⑫ new operator
- ⑬ [ ] operators
- ⑭ Operator precedence
- ⑮ Evaluation Order of operands
- ⑯ new vs newInstance()
- ⑰ instanceof vs isInstanceOf()
- ⑱ ClassNotFoundException vs NoClassDefFoundError

### ① Increment and Decrement operators



Expression	Initial value of x	Value of Y	Final value of x
$y = ++x$	10	11	11
$y = x++;$	10	10	11
$y = --x;$	10	9	9
$y = x--;$	10	10	9

## Case 1

- We can apply increment and decrement operators only for variables but not for constant values. If we are trying to apply for constant values then we will get compiletime error.

Ex 1

```
int x=10;
int y = ++x;
SOP(y); //
```



```
int x=10;
int y = ++10;
```

```
SOP(y); // C-E
```



expected type  
found : value  
required : variable

(2)

- wrapping of increment and decrement operators not allowed

Ex 2

```
int x=10;
int y = ++(++x); // C-E
```

```
SOP(y);
```

C-E

unexpected type  
required: variable  
found: value

(3)

- for final variables we can't apply increment and decrement operators

Ex 3

```
final int x=10;
```

```
x=11;
SOP(x);
```

C-E

can not assign a value to  
final variable x

```
final int x=10;
```

```
x++;
SOP(x);
```

C-E

can not assign a value to  
final variable x

(4)

- we can apply primitive type increment and decrement operators for every except boolean

Ex 4

```
int x=10;
x++;
SOP(x); //
```



```
char ch='a';
```

```
ch++;
SOP(ch);
```

```
'b'
```



```
double d=10.5;
```

```
d++;
SOP(d);
```

```
11.5
```



boolean b=true;

```
b++;
SOP(b);
```

C-E operator ++  
cannot be applied to  
boolean

(5) Difference b/w  $b++$  and  $b = b + 1$

If we apply any arithmetic operator between two variables  $a$  and  $b$ , the result type is always  $\max(\text{int}, \text{type of } a, \text{type of } b)$

Ex: 1:  $\text{byte } a = 10;$

$\text{byte } b = 20;$

$\text{byte } c = \boxed{a+b};$

$\text{Sop}(c);$

C-E: Possible loss of precision  
found & int  
required: byte

$\max(\text{int}, \text{byte}, \text{byte})$

$\text{byte } c = (\text{byte})(a+b); // \text{type casting}$

$\text{Sop}(c); // 30$

Ex: 2:

$\text{byte } b = 10;$

$b = \boxed{b+1};$

$\text{Sop}(b);$

C-E: Possible loss of precision  
found & int  
required: byte

$b = (\text{byte})(b+1);$

Op: 11

Ex:  $\text{byte } b = 10;$

$\boxed{b++};$   
 $\text{Sop}(b); //$

$\max(\text{int}, \text{byte}, \text{int})$

int

$b = (\text{byte})(b+1);$

$\boxed{b++} //$

Ex:  $b = (\text{type of } b)(b+1);$

But in the case of increment & decrement operators internal type casting will be performed automatically.

(2) Arithmetic operators (+, -, \*, /, %)

If we apply any arithmetic operator between two variables  $a$  and  $b$ , the result type is always  $\max(\text{int}, \text{type of } a, \text{type of } b)$

$\text{byte} + \text{byte} = \text{int}$

$\text{byte} + \text{short} = \text{int}$

$\text{short} + \text{short} = \text{int}$

$\text{byte} + \text{long} = \text{long}$

$\text{long} + \text{double} = \text{double}$

$\text{float} + \text{long} = \text{float}$

$\text{char} + \text{char} = \text{int}$

$\text{char} + \text{double} = \text{double}$

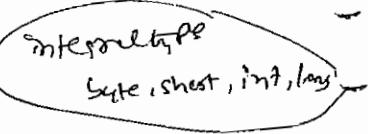
$\text{byte} \rightarrow \text{short} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{double}$

$\text{char}$

Eg:  $\text{sop}('a' + 'b');$   
 $97 + 98$

$\text{sop}('a' + 0.89); \quad 97.89$   
 $97 + 0.89$

AEs (by zero)



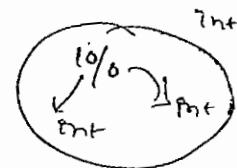
## Infinity

In integral arithmetic (byte, short, int, long) there is no way to represent infinity hence if infinity is result we will get ArithmeticException for integral arithmetic

1) Eg:

$\text{sop}(10/0);$

Re: ArithmeticException: / by zero

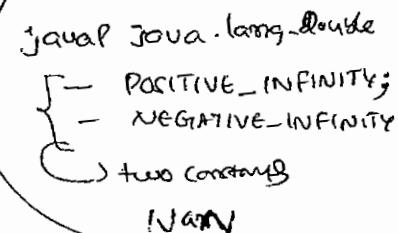


But in floating point arithmetic float and double there is a way to represent infinity for this float & double classes containing the following two constants

POSITIVE\_INFINITY;

NEGATIVE\_INFINITY;

hence Even though result is infinity we won't get any ArithmeticException in floating point Arithmetic



2) Eg:  $\text{sop}(10/0.0); \quad \text{Infinity}$   
 $\text{sop}(-10.0/0); \quad -\text{Infinity}$



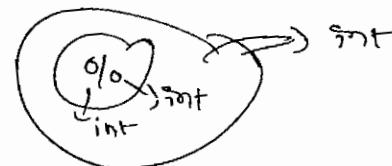
## NAN (Not a Number)

In integral arithmetic (byte, short, int, long)

There is no way to represent undefined results

hence if the result is undefined we will get Runtime exception

Saying ArithmeticException

Eg:  $\text{sop}(0/0); \quad \text{Re: ArithmeticException: / by zero}$ 

0/0 → undefined

But in floating point arithmetic (float & double) there is a way to represent undefined results for this float & double classes containing NaN containing

(61)

hence if the result is undefined we won't get any ArithmeticException  
in floating point Arithmetic

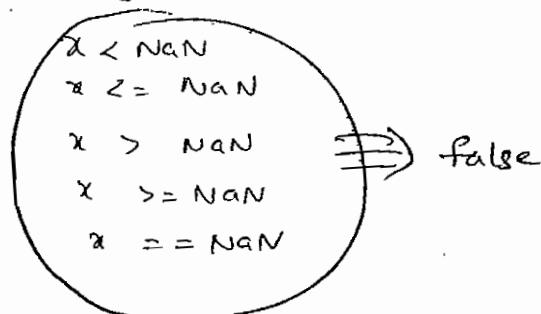
Eg:  $\text{Sop}(0.0/0); \text{NaN}$

$\text{NaN} \rightarrow \text{constant}$

$\text{Sop}(-0/0.0); \text{NaN}$

\* Notes

for any  $x$  value including NaN the following expression returns false



for any  $x$  value including NaN the following expression returns true

$x != \text{NaN} \Rightarrow \text{true}$

Eg:

$\text{Sop}(10 < \text{float.NaN}); \text{false}$

$\text{Sop}(10 \leq \text{float.NaN}); \text{false}$

$\text{Sop}(10 > \text{float.NaN}); \text{false}$

$\text{Sop}(10 \geq \text{float.NaN}); \text{false}$

$\text{Sop}(10 == \text{float.NaN}); \text{false}$

$\text{Sop}(\text{float.NaN} == \text{float.NaN}); \text{false}$

$\text{Sop}(10 != \text{float.NaN}); \text{true}$

$\text{Sop}(\text{float.NaN} != \text{float.NaN}); \text{true}$

### ArithmeticException

1. It is runtimeexception but not compilation error

ArithmeticException

2. It is possible only in integral arithmetic but not in floating point arithmetic

3. The only operators which cause ArithmeticException are / and %.

### ③ String Concatenation operator (+):

The only overloaded operator in Java is + operator.  
Some times it acts as arithmetic addition operator and  
Sometimes it acts as String concatenation operator.

$$\text{Ex: } 10 + 20 = 30$$

$$"ab" + "cd" = abcd$$

for some operator evaluation done from left to right.

→ If atleast one argument is the string type then + operator acts as concatenation operator and if both arguments are number type then + operator acts as arithmetic addition operator.

Ex1:

String a = "durga";

int b=10; c=20; d=30;

SOP (a+b+c+d); durga102030

SOP (b+c+d+a); Godurga

SOP (b+c+a+d); 30durga30

SOP (b+a+c+d); 10durga2030

a+b+c+d

"durga" + 10 + d

"durga" + 10 + d

durga102030

Ex2: Consider the following declarations

String a = "durga";

int b=10, c=20, d=30;  
which of the following expressions are valid.

X ① a + b + c + d;

✓ ② a = a + b + c;

X ③ b = a + c + d;

✓ ④ b = b + c + d;

cc incompatible type  
found: int  
required: g.l. String

C-t: incompatible type  
found: g.lang.String  
required: int

## ⑦ Relational Operators ( $<$ , $\leq$ , $>$ , $\geq$ )

- ① We can apply relational operators for every primitive types except boolean

Eg: `Sop( 10 < 20); true`

`Sop( 'a' < 10); false`

`Sop( 'a' < 97.0); true`

`Sop( 'a' > 'A'); true`

`Sop( true > false)`

Operator  $>$  can not be applied to boolean, boolean

- ② We can't apply relational operators for object types

Eg: `Sop( "dogaliz" > "doga");`

Student, > Student

- ③ Nesting of Relational operators is not allowed  
Otherwise we will get compiletime error

Eg: `Sop( 10 < 20 < 30)`

`true < 30`

Operator  $<$  can not be applied to boolean

int

## ⑤ Equality Operators ( $==$ , $!=$ )

- ① We can apply equality operators for every primitive types including boolean type also

Eg: `Sop( 10 == 20) false`

`Sop( 'a' == 'b'); false`

`Sop( 'a' == 97.0); true`

`Sop( false == false); true`

- ② We can apply equality operators for Object types also  
for Object references  $R_1$ ,  $R_2$

$R_1 == R_2$  returns true if and only if both references pointing to the same object (reference comparison or address comparison)



Ex 1:

```

Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
✓ System.out.println(t1 == t2); false
✓ System.out.println(t1 == t3); true

```



$O \rightarrow t$   
Parent child  
 $O \rightarrow s$

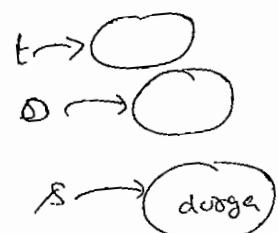
\* if we apply equality operators for Object types then Compulsory there should be some relation b/w argument types (either  $t_1$  child to parent or parent to child or same type) otherwise we will get compiletime error saying Incomparable types & java.lang.String cannot be converted.

Ex 2:

```

Thread t = new Thread();
Object o = new Object();
String s = new String("durga");
✓ System.out.println(t == o); false
✓ System.out.println(o == s); false
✗ System.out.println(s == t);

```



C-E:

Incomparable types & java.lang.String and  
java.lang.Thread

(Q) Difference b/w == operator & .equals() method?

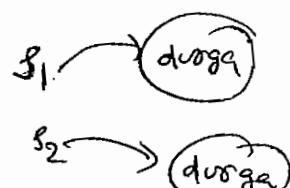
→ In General we can use == operator for reference comparison (address comparison) and .equals() method for content comparison

Ex 3:

```

String s1 = new String("durga");
String s2 = new String("durga");
System.out.println(s1 == s2); false
System.out.println(s1.equals(s2)); true

```



Note

for any object reference  $r$ ,

$r == null$  is always false

$r == null \Rightarrow$  false

but  $null == null$  is always true

$null == null \Rightarrow$  true

String s = null (65)  
 ↴ object reference  
 but internally  
 not pointing  
 so it is  
 null

Eg:

```
String s = new String("durga");
System.out.println(s == null); false
System.out.println(null == null); true
```

```
String s = null;
System.out.println(s == null); true
```

## ⑥ instanceof Operator

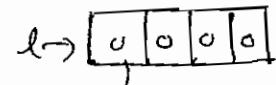
We can use instanceof operator to check whether the given object is of particular type or not.

Object o = l.get(0);  
 Parent reference used to hold child obj

Ex:

```
Object o = l.get(0);
if (o instanceof Student)
{
    Student s = (Student)o;
    // Perform Student Specific functionality
}
else if (o instanceof Customer)
{
    Customer c = (Customer)o;
    // Perform Customer specific functionality
}
```

Allocate new Area



Syntax:

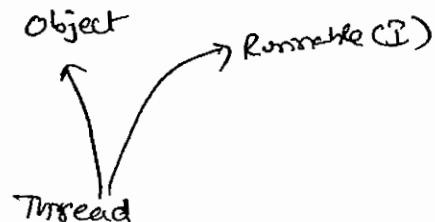
`object instanceof class / interface name`

Object reference

class / interface name

Ex:

```
Thread t = new Thread();
System.out.println(t instanceof Thread); true
System.out.println(t instanceof Object); true
System.out.println(t instanceof Runnable); true
```



Eg 2 To use instanceof operator compulsorily there should be some relation b/w argument types (either child to parent or parent to child or same type) otherwise we will get compilation error saying Inconvertible types.

Eg: 2)

Thread t = new Thread();

sop(t instanceof String); → ~~Compile Error~~

Inconvertible types

found: java.lang.Thread

required: java.lang.String

(3) Note

for any class or interface X null instanceof X is always false

Eg:

sop(null instanceof Thread); false

sop(null instanceof Runnable); false.

## (7) Bitwise operators (&, |, ^)

& → AND ⇒ Returns true iff both arguments are true

| → OR ⇒ Returns true iff at least one argument is true

^ → XOR ⇒ Returns true iff both arguments are different

Eg:

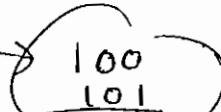
sop(true & false); false

sop(true | false); true

sop(true ^ false); true

\* We can apply these operators for integral types also.

Eg:

sop(4 & 5); 4 → 

sop(4 | 5); 5 → 

sop(4 ^ 5); 1 → 

$\sim$  (tilde)

### Bitwise Complement operator ( $\sim$ )

We can apply this operator only for integral types but not for boolean type.  
If we are trying to apply for boolean type then we will get compilation errors.

Eg:  $sop(\sim \text{true});$  →

$sop(\sim 4); -5$

(C.E: Operator  $\sim$  can not be applied to boolean)

Notes:

The most significant bit acts as sign bit

0 means positive number

1 means negative number

True numbers will be represented directly in the memory whereas negative(-ve) numbers will be represented indirectly in the memory in 2's complement form.

-ve

$4 = 0000 \dots 0100$

Sign bit

$\sim 4 = 1111 \dots 1011$

↓  
2's complement

$000 \dots 0100$  (Anska 12+1)

$00 \dots 0101$

### Boolean Complement operator (!)

We can apply this operator only for boolean types but not for integral types.

Eg:

$sop(!4);$  →

$sop(!\text{false});$

(C.E: Operator ! can not be applied to int)

↓ true



Applicable for both  
boolean & integral types



Applicable for only integral types  
but not for boolean type



Applicable only for boolean  
but not for integral types.

## 8) short-circuit operators (88 ||)

These are exactly same as  
the following differences

bitwise operators (&, |) except

8, |

88, ||

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>① Both arguments should be evaluated always</li> <li>② Relatively performance is low</li> <li>③ Applicable for both boolean and integral types</li> </ul> | <ul style="list-style-type: none"> <li>① second argument evaluation is optional</li> <li>② Relatively Performance is high</li> <li>③ Applicable only for boolean but not for integral types.</li> </ul> |
|--|---|

### Note 8

①  $x \& y \Rightarrow y$  will be evaluated  
If  $x$  is true i.e if  $x$  is  
false then  $y$  won't be evaluated

### Note 28

$x || y \Rightarrow y$  will be evaluated  
If  $x$  is false i.e if  $x$  is true  
then  $y$  won't be evaluated

### Ex 1

```
int x=10, y=15;
if (++x < 10) {  
    ++y > 15  
}  
    x++;  
else  
    y++;  
so (x + " -- " + y);
```

	x	y
8	11	17
88	11	16
1	12	16
11	12	16

### Ex 2

```
int x=10;
if (++x < 10 88 (x/0 > 10))
{
    sop ("Hello");
}
else
    sop ("Hi");
```

① CE

② Re: AE is 1 by zero

③ Hello

④ Hi

if we replace 88 with 8 then we will get Runtimeexception  
saying ArithmeticException / by zero.

## 9) Type-cast operators

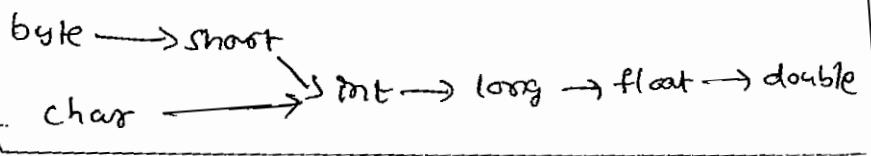
These are two types of type-casting

- ① Implicit type-casting
- ② Explicit type-casting

### ① Implicit type-casting

- ① Compiler is responsible to perform implicit typecasting
- ② whenever we are assigning smaller datatype value to bigger datatype variable implicit type-casting will be performed.
- ③ It is also known as widening or up-casting
- ④ ~~Note~~ There is no loss of information in this type-casting.

The following are various possible conversions where implicit type-casting will be performed.



Ex1:

```
int x = 'a';
```

(Compiler converts char to int automatically by implicit type-casting)

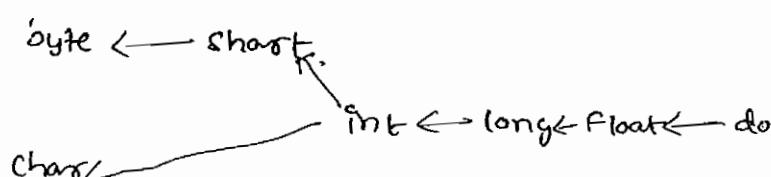
```
System.out.println(x); // 97
```

Ex2:  
double d = 10;  
(Compiler converts int to double automatically by implicit type-casting)

### ② Explicit type-casting

- ① Programmer is responsible to perform explicit typecasting
- ② whenever we are assigning bigger datatype value to smaller datatype variable then explicit type-casting ~~will~~ be required
- ③ It is also known as narrowing or down casting
- ④ There may be a chance of loss of information in this type-casting

The following are various possibilities where explicit type-casting is required



Left → Right ⇒ Implicit-type casting

Right → Left ⇒ Explicit-type casting.

Eg8

```
int x=130;
byte b=a; → C-88
byte b=(byte)x;
    ||
    ||
SOP(b); -126
```

Possible loss of precision  
found: int  
required: byte

Signbit  
 $\begin{cases} 1 \rightarrow \text{-ve} \\ 0 \rightarrow \text{+ve} \end{cases}$

8 bits

Whenever we are assigning bigger datatype value to smaller datatype variable by explicit typecasting the most significant bits will be lost we have to consider only lsb (least significant bits)

Eg8

int x=130  $\Rightarrow$  0000...01000010

byte b=(byte)x  $\Rightarrow$  00000010 → b14

$\begin{array}{r} 130 \\ 2 | 65-0 \\ 2 | 32-1 \\ 2 | 16-0 \\ 2 | 8-0 \\ 2 | 4-0 \\ 2 | 2-0 \\ 1 \end{array}$

Ex2e

```
int x=150;
short s=(short)x;
SOP(s); 150
byte b=(byte)x;
SOP(b); -106
```

2's complement

111101

$$\begin{array}{r} & 1 \\ \hline 111110 & 0 \\ \hline 2^6 2^5 2^4 2^3 2^2 2^1 2^0 \end{array}$$

$$\Rightarrow 64 + 32 + 16 + 8 + 4 + 2 + 1 \Rightarrow 126$$

int x=150; =

$\Rightarrow 000...010010110$

short s=(short)x;  $\Rightarrow$  000...010010110

+ve

$+ (+ve) \Rightarrow$  write as same = 150

byte b=(byte)x;  $\Rightarrow$

$\begin{array}{c} 1 \\ \hline 0010110 \end{array}$

1101001

$$\begin{array}{r} 1101010 \\ \hline 2^6 2^5 2^4 2^3 2^2 2^1 2^0 \end{array}$$

$$64 + 32 + 16 + 8 + 2$$

$\begin{array}{r} 150 \\ 2 | 75-0 \\ 2 | 37-1 \\ 2 | 18-1 \\ 2 | 9-0 \\ 2 | 4-1 \\ 2 | 2-0 \\ 1 \end{array}$

Ex2e double d=130.456;

int x=(float)d;

SOP(x); 130

byte b=(byte)d;

SOP(d); -126

byte res=SOP(d);

Ex3e  
 3) If we assign floating point values to the integral types by explicit typecasting the digits after the decimal point will be lost!

## (16) Assignment operators

There are 3 types of assignment operators

(1) Simple assignment:

Eg: `int x=10;`

(2) Chained assignment:

assignment:

`int a,b,c,d;`

`a=b=c=d=20;`

SOP (`a + "..." + b + "..." + c + "..." + d`);  
 do            do            do            do

We can't

Ex:

Perform chained assignment directly at the time of declaration.

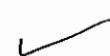
`int a = b = c = d = 20;`

C-E: Can not find symbol  
 symbol: variable b  
 location: class test

Ex 2:

`int b,c,d;`

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



(3) Compound assignment operators:

Sometimes assignment operator mixed with some other operator such type of assignment operators are called compound assignment operators.

Ex:

`int a=10;`

`a+=20;`

`SOP(a); // 30`

The following are all possible compound assignment operators in java.

<code>+ =</code>	<code>&amp; =</code>	<code>&gt;&gt; =</code>
<code>- =</code>	<code>  =</code>	<code>&gt;&gt;&gt; =</code>
<code>* =</code>	<code>^ =</code>	<code>&lt;&lt; =</code>
<code>/ =</code>		
<code>% =</code>		

`>> =` Left shifting assignment operator (6)  
`unsinged rightshift operator`

`x >> 2`

`101011`

`shift left  $\Rightarrow$  >>`

`only with zero  $\Rightarrow$  >>>`

\* In the case of compound assignment operators internal type-casting will be performed automatically

Ex: `byte b=10;`

`b=b+1;`

`SOP(b);`

`(max(int, byte, int))`

$\Rightarrow \text{int}$

C.E.G

Possible loss of precision

found : int

required : byte

byte b=10;

`b++;`

`SOP(b);`

`b=(byte) b+1;`

byte b=10;

`b+=1;`

`SOP(b);`

`b=(byte) (b+1);`

Ex: 2 `int a, b, c, d;`

`a=b=c=d=20;`

~~a=b~~ `a+=b-=c*=d /=2;`

`SOP(a+---" +b+---" +c+---" +d);`

-160

-180

200

10

(11)

Conditional operator (`? :`)

The only possible ternary operator in Java is conditional operator

~~Eg:~~ Syntax:

`int x = (10 < 20) ? 30 : 40;`

We can perform nesting of

conditional operator also

`SOP(x); 30 ✓`

$\rightarrow$  `int x = (10 > 20) ? 30 : ((40 > 50) ? 60 : 70);`

`SOP(x); 70 ✓`

(12)

new operator :

We can use new operator to create object

Ex: `Test t = new Test();`

Note:

① After creating an object constructor will be executed to perform initialization of an object hence constructor is not for creation of object and it is for initialization of an object.

- (2) In Java we have only new keyword but not delete keyword because destruction of useless objects is the responsibility of Garbage collector.

### (13) [ ] Operator :

We can use this operator to declare and create arrays

Eg: int [] x = new int [10];

### (14) Java Operators precedence :

#### 1) Unary operators:

[ ], x++, x--  
++x, --x, ~, !  
new, <type>

#### 2) Arithmetic operators:

\* , /, %  
+, -

#### 3) Shift operators:

>>, >>>, <<

#### 4) ~~Comparison~~ Operators:

<, <=, >, >=, instanceof

#### 5) Equality operators

==, !=

#### 6) Bitwise operators

&  
^  
|

#### 7) Short circuit operators

||

#### 8) Conditional operators

? :

#### 9) Assignment operators

=, +=, -=, \*=, ....

## (15) Evaluation Order of Java Operands

In Java we have only operator precedence but not operand precedence. Before applying any operator all operands will be evaluated from left to right.

Ex:

```
class Test
```

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

```
{        System.out.println(m1(1) + m1(2) * m1(3) / m1(4) + m1(5) * m1(6));
```

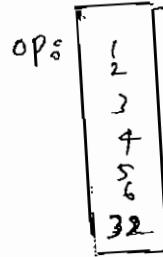
}

```
public static int m1(int i)
```

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

```
    return i;
```

} }



$$1 + 2 * 3 / 4 + 5 * 6$$

$$1 + 6 / 4 + 5 * 6$$

$$1 + 1 + 5 * 6$$

$$1 + 1 + 30$$

$$2 + 30$$

$$\Rightarrow 32$$

4) 6 (1)  
②

## (16) new vs newinstance()

- We can use new operator to create an object if we know class name at the beginning.

Ex:

```
Test t=new Test();
```

```
Student s=new Student();
```

```
Customer c=new Customer();
```

- newinstance() is a method present in class Class

We can use newinstance() method to create object if we don't know class name at the beginning and it is available dynamically at runtime.

Ex:

```
class Student
```

```
{
```

```
}
```

```
class Customer
```

```
{
```

```
}
```

```
class Test
```

```
{    public void main(String[] args) throws Exception
```

```
{        Object o=Class.forName(args[0]).newInstance();
```

↳ class Class

```

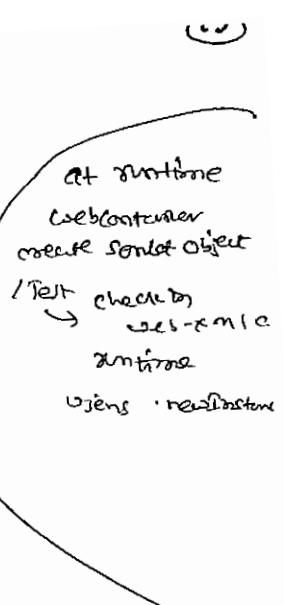
        System.out.println("Object created for "+o.getClass().getName());
    }

    } // Java Test Student
    o/p: object created for : Student.

    Java Test customer {
    o/p: Object created for : Customer

    Java Test java.lang.String {
    o/p: Object created for : java.lang.String

```



→ In the case of new operator based on our requirement we can invoke any constructor

Ex:

```

Test t=new Test();
Test t1=new Test(10);
Test t2=new Test("durga");

```

But newInstance() method internally calls no-argument constructor hence to use newInstance() method compulsorily corresponding class should contain no-argument constructor otherwise we will get RuntimeException exception

\* while using new operator at runtime if the corresponding class file is not available then we will get runtime exception saying NoClassDefFoundError

Ex: Test t=new Test();

at runtime if Test.class file not available then we will get runtime exception saying NoClassDefFoundError: Test

\* while using newInstance() method at runtime if the corresponding class file not available then we will get RuntimeException saying ClassNotFoundException.

Ex:

Object o=Class.forName("aries[0]").newInstance();

Java Test Test123 {
at runtime if Test123.class file is not available then we will get RuntimeException saying ClassNotFoundException: Test123

every servlet class should contain no-arg method  
WebContainer calls newInstance() method

## Differences b/w new and newinstance()

new	newinstance()
① It is an operator in java	① It is a method present in java.lang.class names
② we can use new operator to create object if we know class name at the begining	② we can use newinstance() method to create object if we don't know class name at the begining and it is available dynamically at runtime
③ To use new operator class not required to contain no-argument constructor	③ To use newinstance() method Compulsory class should contain no-argument constructor otherwise we will get Runtime exception saying InstantiationException
④ At runtime if the corresponding class file not available then we will get Runtime exception saying <u>NoClassDefFoundError</u> , which is unchecked	④ At runtime if the corresponding class file not available then we will get RuntimeException saying ClassNotFoundException, which is checked

### Q8) Difference b/w ClassNotFoundException and NoClassDefFoundError

for hardcoded class names, at runtime if the corresponding .class file is not available then we will get RuntimeException saying NoClassDefFoundError, which is unchecked.

Eg:

```
Test t = new Test();
```

At runtime if Test.class file is not available then we will get RuntimeException saying NoClassDefFoundError : Test

for dynamically provided class names at runtime if the corresponding class file not available then we will get RuntimeException saying ClassNotFoundException, which is checked exception

Eg:

```
Object o = Class.forName("arg[0]").newInstance();
```

java Test student

At Runtime if Student.class file not available then we will get RuntimeException saying ClassNotFoundException : Student

## (17) instanceof vs isInstance()

- ① instanceof is an operator in java we can use instanceof to check whether the given object is of particular type or not and we know the type at the beginning

Ex:

```
Thread t = new Thread();
System.out.println(t instanceof Runnable);
System.out.println(t instanceof Object);
```

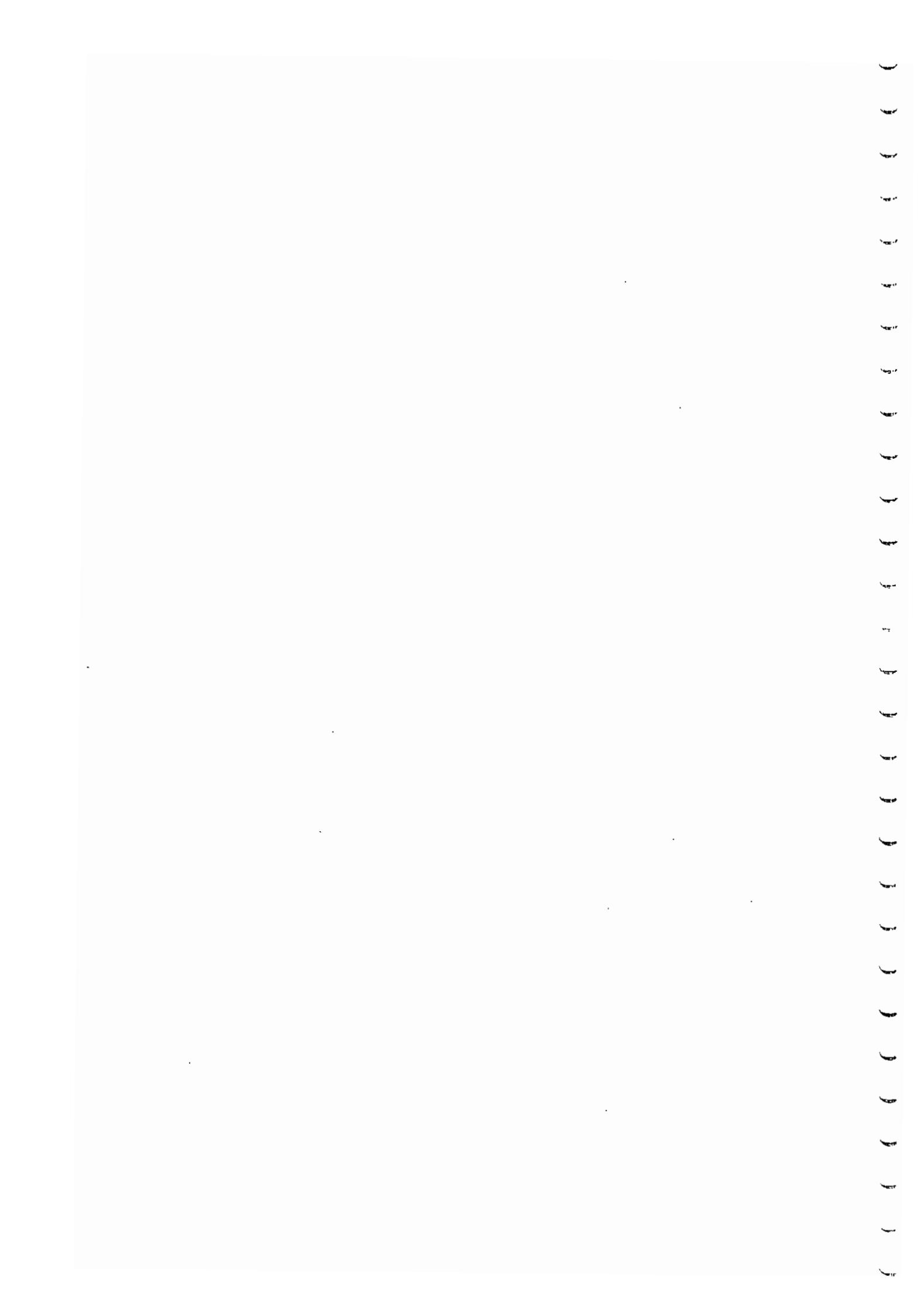
- ② isInstance() is a method present in `java.lang.Class` we can use isInstance() method to check whether the given object is of particular type or not and we don't know the type at the beginning and it is available dynamically at runtime

Ex:

```
class Test
{
    public static void main(String[] args) throws Exception
    {
        Thread t = new Thread();
        System.out.println(Class.forName(args[0]).isInstance(t));
    }
}
```

```
Java Test o/p: Runnable <
Java Test String <
o/p: false.
```

→ isInstance() method is method equivalent of instanceof operator.



## flow-control

flow-control describes the order in which the statements will be executed at runtime.

### flow-control

#### 1. Selection Statements

- ① if - else
- ② switch()

#### 2. Iterative Statements

- ① while()
- ② do-while()
- ③ for()
- \* ④ for-each loop

#### 3. Transfer Statements

- ① break
- ② continue
- ③ return
- ④ try-catch-finally
- ⑤ assert

#### Selection Statements

##### ① if - else

Syntaxe → should be boolean type  
`if (b)  
{  
 Action if b is true  
}  
else  
{  
 Action if b is false  
}`

The argument to the if statement should be boolean type by mistake if we are trying to provide any other type then we will get compiletime error.

Ex:

① `int x=10;  
if(x)  
{  
 sop ("Hello");  
}  
else  
{  
 sop ("Hi");  
}`

→ **Incompatible types**  
**found = int**  
**required: boolean**

② `int x=10;  
if (x == 20)  
{  
 sop ("Hello");  
}  
else  
{  
 sop ("Hi");  
}`

③ `int x=10;  
if (x == 20)  
{  
 sop ("Hello");  
}  
else  
{  
 sop ("Hi");  
}`

OPG: **Hi**

- ↑ ① Selection, ② Iterative, ③ Transfer Statement
1. among several options select only one option
  2. Group of statements created iteratively
  3. Transfer control from one place to another

```

④ boolean b=true;
if (b==false)
{
    System.out.println("Hello");
}
else
{
    System.out.println("Hi");
}
System.out.println("Hello");

```

```

⑤ boolean b=false;
if (b==false)
{
    System.out.println("Hello");
}
else
{
    System.out.println("Hi");
}
System.out.println("Hello");

```

### Conclusion 2

else part and curly braces are optional without curly braces only one statement is allowed under if which should not be declarative statement.

```

if (true)
    System.out.println("Hello");

```

```

if (true);

```

```

if (true)
    int a=10;

```

```

if (true)
{
    int a=10;
}

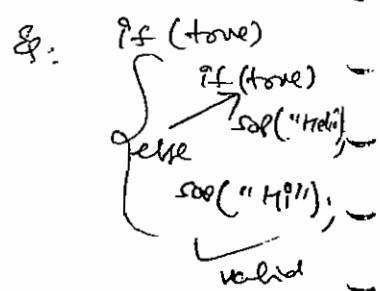
```

### Notes

Semicolon (;) is a valid java statement which is also known as empty statement.



Note: There is no dangling else problem in java. Every else is mapped to the nearest if statement.



## ② Switch Statement

If several options are available then it is not recommended to use nested if-else because it reduces readability to handle this requirement we should go for switch statement.

### Syntax:

Switch (X)

```

    {
        case1: Action-1;
        break;

        case2: Action-2;
        !
        break;

        caseN: Action-n;
        break;

        default: default Action
    }
  
```

boolean -  
true/false  
only else

long →  
-2147483648 to  
2147483647 → (long)

float 0 to  
double 0.1, 0.11... &  
∞ cases  
1.5 →  
Auto boxing case

### Cases

① The allowed argument types for the switch statement are

byte  
short  
char  
int } until 1.4 version but from 1.5 version onwards corresponding wrapper classes and enum type also allowed. from 1.7 version onwards String type also allowed.

1.4V	1.5V	1.7V
byte	Byte	
short	short	
char	Character	String
int	Integer + enum	

② curly braces are mandatory except switch everywhere curly braces are optional

Eg: Switch (X)

{  
}

③ Both case and default are optional

ie an empty switch statement is a valid java syntax

Eg:  
int x=10;  
switch(x)  
{  
}

③ Inside switch every statement should be underscore case or default i.e independent statements are not allowed inside switch otherwise we will get compile time error.

Ex:

```
int x=10;  
switch(x)  
{  
    sop("Hello");  
}
```

C.eg

case, default or } expected

④ Every case table should be compiletime constant (i.e constant expression)

Ex:

```
int x=10;  
int y=20;  
switch(x)  
{  
    case 10:  
        sop(10);  
        break;  
    case y:  
        sop(20);  
        break;  
}
```

C.eg

constant expression required

\* If we declare error.

y as final then we won't get any compiletime

⑤ Both switch argument case table should be and case label can be expressions but constant expression

Ex:

```
int x=10;  
switch(x+1) {  
    case 10:  
        sop(10);  
        break;
```

valid

case 10+20+30:

sop(60);

⑥

Every case table should be in the range of switch argument type otherwise we will get compiletime error.

Ex:

```

byte b=10;
switch(b)
{
    case 10: sop(10);
                break;
    case 100: sop(100);
                break;
    case 1000: sop(1000);
}

```

Case 1000:

Possible loss of precision  
 - found & int  
 - required = byte  
 byte range → 127 to 128

```

byte b = 10;
switch(b+1)
{
    case 10: sop(10);
                break;
    case 100: sop(100);
                break;
    case 1000: sop(1000);
}

```

- ⑦ Duplicate case labels are not allowed otherwise code will get compilation error

Skt

```

int x=10;
switch(x)
{
    case 97: sop(97);
                break;
    case 98: sop(98);
                break;
    case 99: sop(99);
                break;
    case 'a': sop('a');
}

```

Case Duplicate Case Label

- Case Label
- 1. It should be constant expression
  - 2. The value should be in the range of switch argument type
  - 3. Duplicate Case Labels are not allowed.

## fall-through inside switch

With in the switch if any case is matched from that case onwards all statements will be executed until break or end of the switch this is called fall-through inside switch

The main advantage of fall-through inside switch is we can define common action for multiple cases (code reusability)

Ex:

Switch(x)

{

Case 1:

Case 2:

Case 3:

SOP("Q-4");                  Quarter  
break;

Case 4:

Case 5:

Case 6: SOP("Q-1");  
break

}

||

x=0

0

x=1

1

x=2

2

x=3

def

def

Ex:

Switch(x)

{

Case 0: SOP(0);

Case 1: SOP(1);  
break;

Case 2: SOP(2);

default: SOP("default");

}

## default case

- 1) With in the switch we can take default case atmost once
- 2) default case will be executed iff (if and only if) there is no case matched
- 3) With in the switch we can write default case anywhere but it is recommended to write as last case

Ex:

Switch(x)

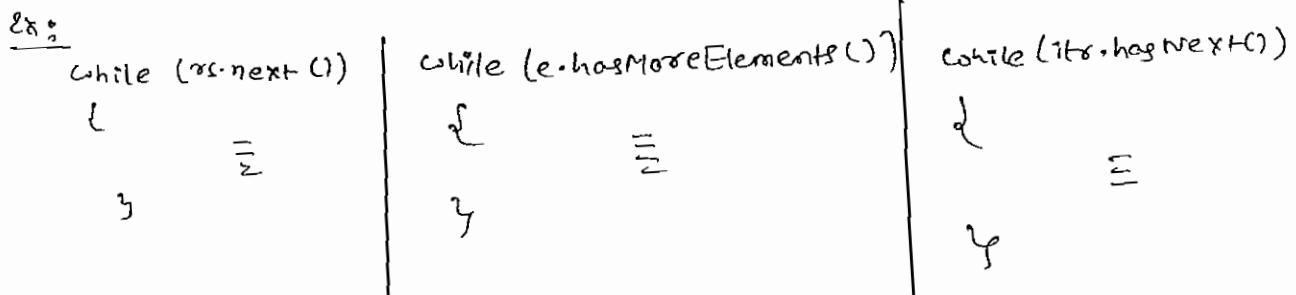
{

default: SOP("def");  
  
 Case 0: SOP(0);  
break;  
  
 Case 1: SOP(1);

	<u>Case 2: SOP(2);</u>	<u>x=0</u>	<u>x=1</u>
{		0	1
		2	def
		3	def

## (2) Iterative Statements

① while if we don't know number of iterations in advance then we should go for while loop.



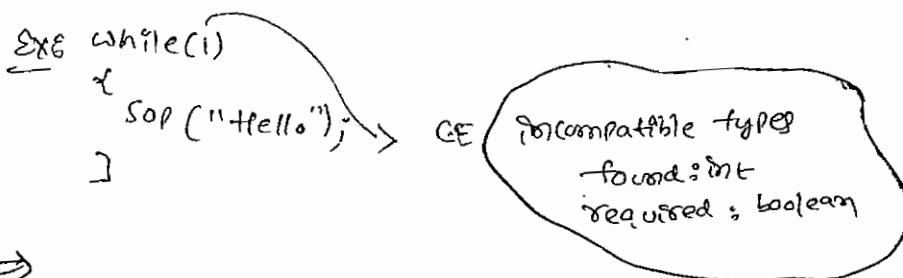
### Syntax

`while (b)`

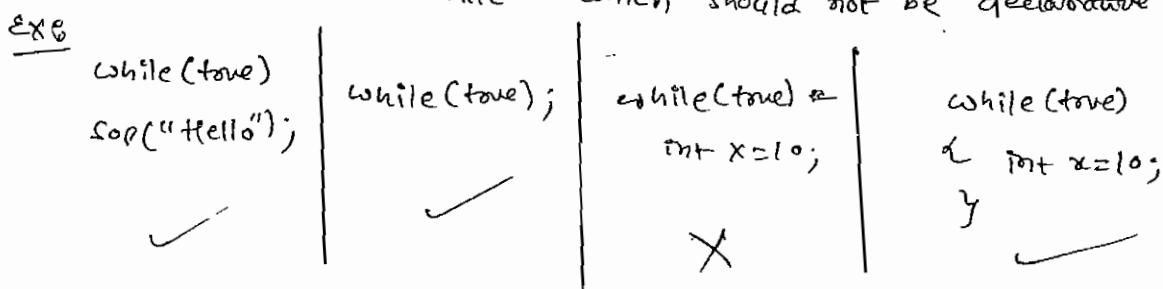
↳ Action

↳ should be boolean type

The argument should be boolean type if we are trying to provide any other type then we will get compiletime error



Curly braces are optional and one statement under while without curly braces we can take only which should not be declarative statement.



a Value - 10  
b value - 20  
final int a=10, b=20  
so compiler aware

white(true) → executed atleast 20 times  
number of times  
downwhile → atleast 1 time.

10 < 20  
true keep on executing

so ↓  
`System.out.println("Hi")` will not get  
so unreachable statement

int a=10;  
while(a>b)  
& `System.out.println("Hello");`  
& `System.out.println("Hi")`

here a is b values are  
variables changed at  
time so compilers will  
check, JVM only exec

`final int a=10, b=20;`  
`while(a>b)`

final value declared for constants

① `while (true)`  
`    ` `sop ("Hello");`  
`    ` `y`  
`    ` `sop ("Hi");`  
`  
`-e: on reachable statement X

② `while (false)`  
`    ` `sop ("Hello");`  
`    ` `y`  
`    ` `sop ("Hi");`  
`  
`-e: unreachable statement X  
`

③ `int a=10, b=20`

`while (a < b)`  
`    ` `sop ("Hello");`  
`    ` `y`  
`    ` `sop ("Hi");`

o/p:

Hello.

Hello

:

④ `int a=10, b=20`  
`    ` `while (a > b)`  
`    ` `sop ("Hello");`  
`    ` `y`  
`    ` `sop ("Hi");`  
`  
`  
` o/p: Hi

⑤ `final int a=10, b=20;`  
`    ` `while (a < b) → true`  
`    ` `{`  
`    ` `sop ("Hello");`  
`    ` `} sop ("Hi");`  
`  
`-e: unreachable stmt

10<20  
true

⑥ `final int a=10, b=20;`  
`    ` `while (a > b) → false`  
`    ` `{`  
`    ` `sop ("Hello");`  
`    ` `} sop ("Hi");`  
`  
`-e: unreachable stmt

Note  
→ ①

Every final variable will be replaced by the value at compiletime only

Ex: `final int a=10;`  
`    ` `int b=20;`  
`    ` `sop(a); } After compilation → sop(10);`  
`    ` `sop(b); } = compilation → sop(20);`

Note 2: If every argument is a final variable (compiletime constant) then that operation should be performed at compiletime only

Ex: 2 `final int a=10, b=20;`  
`    ` `int c=20;`

`sop (a+b);`  
`    ` `sop (a+c); } After compilation → sop (30);`  
`    ` `sop (a < b);`  
`    ` `sop (a < c);`

`sop (30);`  
`    ` `sop (10+c);`  
`    ` `sop (true);`  
`    ` `sop (10 < c);`

## ② do-while()

If we want to execute loop body atleast once then we should go for do-while()

Syntax:

```
do
{
    body
}
while (b);
```

Mandatory  
should be  
boolean type

→ curly braces are optional and without curly braces we can take only one statement b/w do and while which should not be declarative statements.

① do  
 sop("Hello");  
 while (true);

② do;  
 while (true);

③ do  
 int a=10;  
 while (true);

④ do  
 ~~int a=10;~~  
 while (true);

⑤ do  
 while (true);  
 (→ without body  
 what is the use  
 so invalid)

Explain  
do while (true)  
sop ("Hello");  
while (false);

do  
while (true)  
sop ("Hello");  
while (false);

- {  
1) Considering one  
start  
2) after while;  
X ⑥ not unreachable  
already entered

① do  
 { sop ("Hello");  
 } while (true);  
 sop ("Hi");  
 C.E: unreachable  
 statement

② do  
 { sop ("Hello");  
 } while (false);  
 sop ("Hi");  
 Output  
 Hello  
 Hi

③ `int a=10, b=20;`

do

{  
  `sop ("Hello");`

}  
  `while (a < b);` normal  
    `sop ("Hi");` variable

Compiler will not check  
no true or false  
JVM will not responsible

op:  
Hello  
Hello  
Hello  
}

④ `final int a=10, b=20;`

do

{  
  `sop ("Hello");`

}  
  `while (a < b);`

`sop ("Hi");`

CSE unreachable statement

⑤

`int a=10, b=20;`

do

{  
  `sop ("Hello");`

}  
  `while (a > b);` normal  
    `sop ("Hi");` variable

⑥ `final int a=10, b=20;`

do

{  
  `sop ("Hello");`

}  
  `while (a > b);`

`sop ("Hi");`

op: Hello  
Hi

③ for loop

① for loop is the most commonly used loop in Java

② if we know numbers of iterations in advance then for loop is the best choice.

Structure

`for (initialization-section;  
      conditional-check;  
      increment-decrement-section)`

{

loop body;  
  ③, ⑥, ⑨

}

①

②, ⑤, ⑧

④, ⑦

curly braces are optional and without curly braces we can take only one statement under for loop, which should not be declarative statement

`for (int i=0; true; i++)  
  sop ("Hello");`

✓

`for (int i=0; i<10; i++)  
  sop ("Hello");`

→

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

X

Initialization section

- 1) This part will be executed only once in loop lifecycle
- 2) here we can declare and initialize local variables after for loop
- 3) here we can declare any number of variables but should be of the same type by mistake if we are trying to declare different datatype variables then we will get compiletime error

Ex: `int i=0, j=0;` ✓

~~`int i=0, string s="durga";`~~ X  
~~`int i=0; int j=0;`~~ X

- \* In the initialization section we can take any valid java statement including `SOP (System.out.println)`

Ex: `int i=0;  
for (SOP("Hello Boss u R sleeping"); i<3; i++)  
{  
 SOP("No Boss u only sleeping");  
}`

O/P: Hello Boss u R sleeping  
 No Boss u only sleeping  
 No Boss u only sleeping  
 No Boss u only sleeping

Conditional check

- 1) Here we can take any valid boolean expression but should be of the type (`as32&bc2011(c>6)`)
- 2) This part is optional and if we are not taking anything then compiler will always place true

Increment or Decrement Section

In the Increment or Decrement section we can take any valid java statement including `SOP (System.out.println)`

Ex: `int i=0;`

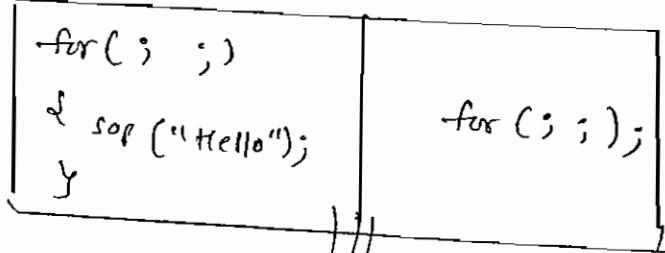
`for (SOP("Hello"); i<3; SOP("Hi"))  
{  
 i++;  
}`

O/P: Hello

Hi  
 Hi  
 Hi

→ All 3 parts of for loop are independent of each other and optional

Ex 5



Infinite loops.

Unreachable

①

```
for (int i=0; true; i++)
{
    sop("Hello");
}
sop("Hi");
C.E.: unreachable
stmt
```

② for (int i=0; false; i++)
{
 sop("Hello");
}
sop("Hi");
{'{'

unreachable stmt

③ for (int i=0; ; i++)
{
 sop("Hello");
}
sop("Hi");
{'{'

C.E.: unreachable stmt

④

```
int a=10, b=20;
for (int i=0; a < b; i++)
{
    sop("Hello");
}
sop("Hi");
O/P:
Hello
Hello
{'{'
```

⑤ int a=10, b=20;
for (int i=0; a > b; i++)
{
 sop("Hello");
}
sop("Hi");
{'{'

⑥ final int a=10, b=20;
for (int i=0; a < b; i++)
{
 sop("Hello");
}
sop("Hi");
{'{'

⑦

```
fixed int a=10, b=20;
for (int i=(0; a > b; i++)
{
    sop("Hello");
}
sop("Hi");
C.E.: unreachable stmt
{'{'
```

## ~~for each loop~~ for-each loop (Enhanced for loop)

- Introduced in 1.5 version
- It is specially designed loop to retrieve elements of Arrays and Collections.

Ex 1: To print elements of one dimensional Array

`int[] x = {10, 20, 30, 40};`

$x \rightarrow [10|20|30|40]$

Normal for loop

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

enhanced for loop

```
for (int x1 : x)
{
    System.out.println(x1);
}
```

Ex 2:

To print elements of two dimensional Array

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

$x \rightarrow \begin{bmatrix} & & \\ & x_1 & \\ \begin{bmatrix} 10 | 20 | 30 \end{bmatrix} & & \begin{bmatrix} 40 | 50 \end{bmatrix} \end{bmatrix}$

Normal for loop

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

enhanced for loop

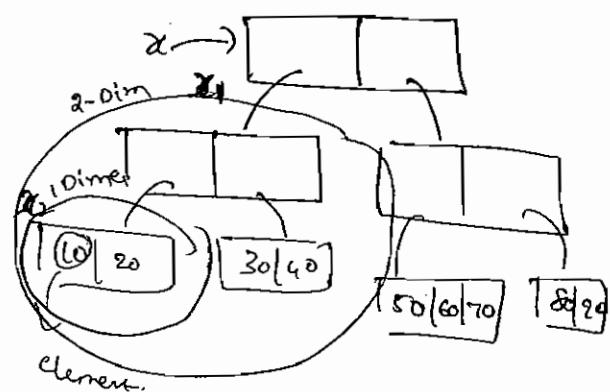
```
for (int[] x1 : x)
{
    for (int x2 : x1)
    {
        System.out.println(x2);
    }
}
```

Ex 3:

To print elements of

Three dimensional Array

```
for (int[][][] x1 : x)
{
    for (int[][] x2 : x1)
    {
        for (int[] x3 : x2)
        {
            System.out.println(x3);
        }
    }
}
```



\* for each loop is the best choice to iterate elements of Array and Collections but it's a limitation is that it's applicable only for Arrays & Collections and it's not a general purpose loop.

Ex:

```
for (int i=0; i<10; i++)
{
    System.out.println("Hello");
}
```

we can't write an equivalent for each loop directly.

\* By using normal for loop we can print array elements either in original order or in reverse order  
But by using for-each loop we can print array elements only in original order but not in reverse order

Ex:

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

```
for (int i=x.length-1; i>=0; i--)
{
    System.out.println(x[i]);
}
```

Output:  
50  
40  
30  
20  
10

we can't write an equivalent for each loop directly

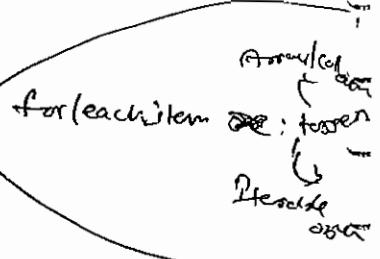
## Iterable (I)

Syntaxe

```
for (each item x : target)
{
    // ...
}
```

↳ IterableObject

↳ Array / Collection



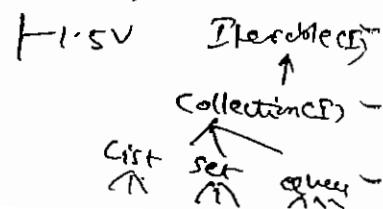
The target element in for-each loop should be iterable object  
an object is said to be iterable iff (if and only if) corresponding class implements java.lang.Iterable (I)

Iterable() Interface  
method

introduced in 1.5 version and it contains only one

Public Iterator iterator()

## Iterator (I)



All Array related classes and collection implemented classes already implemented Iterable (I) interface being a programmer we are not required to do anything just we should aware the point.

differences b/w Iterator and Iterable

### Iterator (I)

- ① It is related to Collections
- ② We can use to retrieve elements of Collections one by one
- ③ Present in java.util package
- ④ It contains 3 methods
  - 1) hasNext()
  - 2) next()
  - 3) remove()

### Iterable (I)

- ① It is related to for-each loop
- ② The target element in for-each loop should be Iterable.
- ③ present in java.lang package
- ④ It contains one method  
iterator()

## Transfer Statements

### break

We can use break statement in the following places

- ① Inside switch To stop fall through

```
Ex: int x=0;
switch(x)
{
    case 0:
        sop(0);
    case 1:
        sop(1);
        break;
    case 2:
        sop(2);
    default:
}
```

sop("def");      Output: 0 1

- ② Inside loops

To break loop execution based on some condition

```
for (int i=0; i<10; i++)
{
    if (i==5)
```

3	break;	0
	sop(i);	1
		2
		3

### 3) override Labeled Block

To break block execution based on some condition

Ex:

```

class Test
{
    public static void main(String[] args)
    {
        int x=10;
        l1:
        {
            System.out.println("begin");
            if (x==0)
                break l1;
            System.out.println("end");
        }
        System.out.println("Hello");
    }
}
  
```

Labeled block  
cent for readability  
DB maintainability

→ These are the only places where we can use break statement.  
If we are using anywhere else we will get compilation error  
saying break outside switch or loop

Ex:

```

class Test
{
    public static void main(String[] args)
    {
        int x=10;
        if (x == 10)
            break;
        System.out.println("Hello");
    }
}
  
```

C.E.

break outside switch or loop

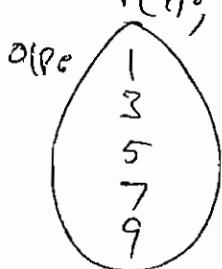
### Continue

We can use continue statement inside loops to skip current iteration and continue for the next iteration

Ex:

```

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



we can use continue statement only inside loops. If we are using anywhere else we will get compiletime error saying

### Continue outside of loop

Ex: Class Test

```

d   P  s v main (String[] args)
{
    int x = 0;
    if (x == 0) {
        continue;
    }
    for ("Hello");
}
  
```

c.e.: continue outside of loop

### Labeled break and continue

We can use labeled break and continue to break or continue a particular loop for nested loops

Ex:

```

l1:
for (----)
{
    l2:
    for (----)
    {
        ;
        ;
        for (----)
        {
            break l1;
            break l2;
            break;
        }
    }
}
  
```

lc

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

break; → ①

Sop(i + " --- " + j);

use instead  
↓ ①

break;  
1---0  
2---0  
2---1

break l1;

No output

Continue;  
0---1  
0---2  
1---0  
#---2  
2---0  
2---1

Continue l1;

i---0  
2---0  
2---1

do-while() vs Continue

[dangerous combination]

```
int x=0;
do
    x---X
    x++;
    Sop(x);
    if (++x < 5)
        Continue;
    x++;
    Sop(x)
} while (++x < 10);
```

1  
4  
6  
8  
10

x=0  
X  
3  
4  
6  
7  
8  
10  
11

Inside do-while → continue will not go for first line until check  
while loop

23/09/14

## Declarations & Access Modifiers

- (1) Java source file structure
- (2) Class Level modifiers
- (3) Member Level modifiers
- (4) Interfaces

### (1) Java source file structure

A java program can contain any number of classes but atmost one class can be declared as public. If there is a public class then name of the program and name of the public class must be matched otherwise we will get compiletime error.

Ex:

```

Class A
{
}
Class B
{
}
Class C
{
}

```

Case 1:

If there is no public class then we can use any name and there are no restrictions.

A.java

B.java

C.java

Burga.java

Case 2:

If class B is public then name of the program should be B.java otherwise we will get compiletime error saying

C.E Class B is public, should be declared in a file named B.java

If class B and C declared as public and name of the program is B.java then we will get compiletime error saying

C.E Class C is public, should be declared in a file named C.java.

Ex-2

```

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

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

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

class D
{
}

```

Durga.java

Java A

javac A class main

Java B

javac B class main

Java C

javac C class main

Java D

javac D class main

javac Durga.java

A.class

B.class

C.class

D.class

java Durga

RE: NoClassDefFoundError: Durga

Conclusion

- ① whenever we are compiling a java program present by the program a separate .class file will be generated.
- ② we can compile a java program (java sourcefile) but we can run a java .class file.

- (3) whenever we are executing a java class the corresponding class main method will be executed.
- If the class doesn't contain main method then we will get runtime exception saying NoSuchMethodError:main
- (4) If the corresponding .class file not available then we will get RuntimeException saying ClassNotFoundException.
- (5) It is not recommended to declare multiple classes in a single source file.  
It is highly recommended to declare only one class per source file.  
and name of the program we have to keep same of class name  
The main advantage of this approach is readability & maintainability of the code will be improved.

### Import statement

```

class Test
{
    public static void main (String[] args)
    {
        ArrayList l = new ArrayList ();
    }
}

```

c.g.: can not find symbol  
 symbol: class ArrayList  
 location: class Test

We can solve this problem by using fully qualified name

java.util.ArrayList l = new java.util.ArrayList ();  
 ↓  
 fully qualified name

- The problem with usage of fully qualified name everytime is it increases length of the code & reduces readability
- We can solve this problem by using import statement
- Whenever we are writing import statement if it is not required to use fully qualified name everytime we can use short name directly

Ex:  
import java.util.ArrayList  
class Test
{
 public static void main (String[] args)
 {
 ArrayList l = new ArrayList ();
 }
}

↓  
 short name

Hence import statement acts as typing shortcut.

25-09-14

### Case 1: Types of import statements

There are 2 types of import statements

- (1) Explicit class import
- (2) Implicit class import

#### Explicit class import:

Eg: import java.util.ArrayList;

→ It is highly recommended to use explicit class import because it improves readability of the code  
Best suitable for high technology where readability is important.

#### Implicit class import:

Eg: import java.util.\*;

→ Not recommended to use because it reduces readability of the code  
Best suitable for ameepet where typing is important.

### Case 2:

Which of the following import statements are meaningful

- ✓ (1) import java.util.ArrayList
- X (2) import java.util.ArrayList.\*;
- ✓ (3) import java.util.\*;
- X (4) import java.util;

### Case 3:

Consider the following code

```
Class myObject extends java.rmi.UnicastRemoteObject
{
    ...
}
```

The code compiles fine even though we are not writing import statement because we use fully qualified name.

Notes

whenever we are using fully qualified name it is not required to write import statement similarly whenever we are writing import statement it is not required to use fully qualified name.

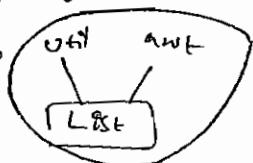
Case 4:

```
import java.util.*;  
import java.sql.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Date d = new Date();  
    }  
}
```

C.E: Reference to Date is ambiguous

Notes

even for the case of List also we may get same ambiguity problem because it is available in both Util & AWT Packages

Case 5:

while resolving class names compiler will always gives the precedence in the following order

- (1) Explicit class import
- (2) classes present in current working directory (default package)
- (3) implicit class import.

Ex:

```
import java.util.Date;  
import java.sql.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Date d = new Date();  
        System.out.println(d.getClass().getName());  
    }  
}
```

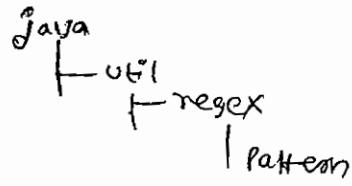
In the above example Util package Date got considered

Case 6:

whenever we are importing a java package all classes and interface present in that package by default available but not subpackage classes. If we want to use subpackage class compulsory we should write import statement until subpackage level.

Ex:

To use Pattern class in our program which import statement is required



- ① import java.\*;
- ② import java.util.\*;
- ③ import java.util.regex.\*;
- ④ No import required

Case 7:

All classes and interfaces present in the following packages are by default available to every Java program hence we are not required to write import statement.

- ① java.lang package
- ② default package (current working directory)

Case 8:

Import statements is totally compilation related concept. If more number of imports then more will be the compilation but there is no effect on execution time (runtime).

Case 9:

Difference b/w C-language #include & Java language import statement

- 1) In the case of C Language #include all input output header files will be loaded at the beginning only (at translation time) hence it is static include.
- 2) But in the case of Java import statement no .class file will be loaded at the beginning whenever we are using a particular class then only corresponding .class file will be loaded this is like dynamic include (or) load on demand (or) load on fly.

Note:

1.5 version new features

- ① for-each loop
- ② var-arg method
- ③ Autoboxing & Auto-unboxing
- ④ Generics
- ⑤ Co-varient return types
- ⑥ Queue
- ⑦ Annotations
- ⑧ enum
- ⑨ static import

## Static import

- Introduced in 1.5 version
  - According SUN usage of static import reduces length of the code and improves readability but according to worldwide program experts (like us) usage of static import creates confusion and reduces readability. hence if there is no specific requirement then it is not recommended to use static import.
- usually we can access static members by using class name but whenever we are writing static import we can access static members directly without class name.

Without static import

```
class Test
{
    public static void main (String [] args)
    {
        System.out.println (Math.sqrt(4));
        System.out.println (Math.max(10,20));
        System.out.println (Math.random());
    }
}
```

With static import.

```
import static java.lang.Math.*;
Class Test
{
    public static void main (String [] args)
    {
        System.out.println (sqrt(4));
        System.out.println (max(10,20));
        System.out.println (random());
    }
}
```

28/09/14

Explain about System.out.println

```
class Test
{
    static String s = "java";
}
```

Static variable  
accessed using  
class name  
Test.s

Test.s.length()

'Test' is  
class Name

's' is a  
static variable  
present in  
Test class of the  
type java.lang.String

.length() is a  
method present in  
String class

class System

{ static PrintStream out;

!!

System.out.println()

System is a class present in java.lang.~~static~~ package

'out' is a static variable of System class

Println() is a method present in PrintStream class.

\* Out is a static variable present in System class hence we can access it by using class name System. But whenever we are writing static import Out directly

System.out.println();  
With out using class name how to access static members, using static import

Ex  
import static java.lang.System.out;  
class Test  
{  
 public static void main(String[] args)  
 {  
 out.println("Hello");  
 out.println("Hi");  
 }  
}

Every number type wrapper class contains MAX-VALUE

import static java.lang.Integer.\*;  
import static java.lang.Byte.\*;  
public class Test  
{  
 public static void main(String[] args)  
 {  
 System.out.println(MAX\_VALUE);  
 }  
}

CSE

Reference to MAX-VALUE is ambiguous

while Resolving static members compiler will always consider the precedence (or) priority in the following order

- ① Current class static members
- ② Explicit static import
- ③ Implicit static import

(1) Highest Priority  
Explicit class  
(2) Import  
Current class  
(3) Implicit class  
Import  
Normal import only

```

import static java.lang.Integer.MAX_VALUE; → ②
import static java.lang.Byte.*;
public class Test {
    static int MAX_VALUE = 999; → ①
    public static void main(String[] args) o/p: 999
        System.out.println(MAX_VALUE);
}

```

If we comment ~~line ②~~ both lines ① & ② then  
will be considered in this case output is

(cos)  
if we comment line ① then explicitly static import will be consider and hence Integer class max value will be consider in this case the output is

8147483647

Implicit static import  
127 (Byte class max-value)

### Normal Import

#### ① Explicit import:

Syntax:

Eg: `import Packagename.classname;`  
`import java.util.ArrayList;`

#### ② Implicit import:

Syntax:

`import Packagename.*;`  
Eg: `import java.util.*;`

### Static Import

#### ① Explicit static import:

Syntax:  
`import static Packagename.classname.static members;`  
Eg: `import static java.lang.Math.sqrt;`

#### ② Implicit static import:

Syntax:  
Eg: `import static Packagename.classname.*;`  
`import static java.lang.Math.*;`  
`import static java.lang.System.*;`

(Q) Which of the following import statements are valid?

import java.lang.Math.\*; X  
import static java.lang.Math.\*; ✓  
import java.lang.Math.sqrt; X  
import static java.lang.Math.sqrt(); X  
import java.lang.Math.sqrt.\*; X  
import static java.lang.Math.sqrt; ✓  
import java.lang; X  
import static java.lang; X  
import java.lang.\*; ✓  
import static java.lang.\*; X

Ans.  
1. . . .

- Two packages containing a class or interface with the same name is very rare and hence ambiguity problem is also very rare in normal import.
- but 2 classes or interfaces containing a variable or method with same name is very common and hence ambiguity problem is also very common problem in static import.
- Usage of static import reduces readability and creates confusion and hence if there is no specific requirement then it is not recommended to use static import.

#### Difference b/w Normal import & static import

- We can use normal import to import classes & interfaces of a particular package.  
Whenever we are using normal import it is not required to use fully qualified name and we can use short names directly.
- We can use static import to import static members of a particular class or interface.  
Whenever we are writing static import it is not required to use class name to access static members and we can access directly.

## Packages

### Package

It is an encapsulation mechanism to group related classes & interfaces into a single unit, which is nothing but package.

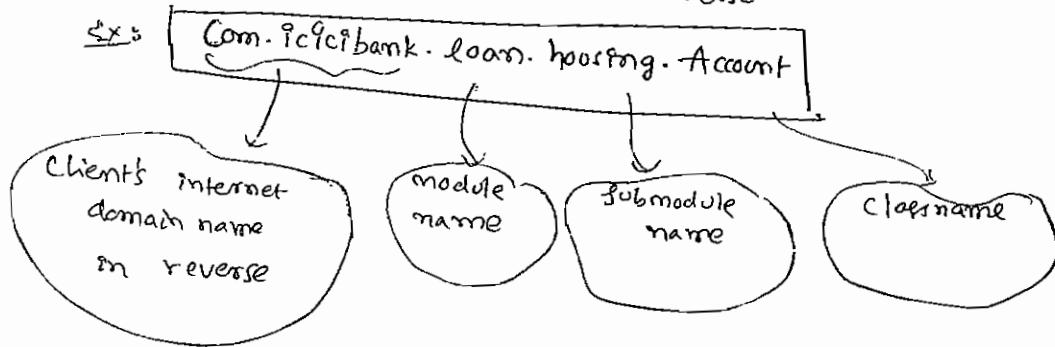
Ex1: All classes & interfaces which are required for Database operations are grouped into a single package which is nothing but java.sql package.

Ex2: All classes & interfaces which are useful for file I/O operations are grouped into a separate package which is nothing but java.io package.

### The main advantages of package are

- ① To resolve naming conflicts i.e unique identification of our components;
- ② It improves modularity of the application
- ③ It improves maintainability of the application
- ④ It provides security for our components

There is one universally accepted naming convention for Packages i.e to use internet domain name in reverse.



```

    ➔ Package com.durgasoft.scjp;
        public class Test
        {
            public static void main(String[] args)
            {
                System.out.println("Pkg demo");
            }
        }
  
```

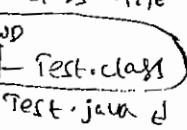
① Javac Test.java ↴

Generated .class file will be placed in

current working directory  
Corresponding package structure

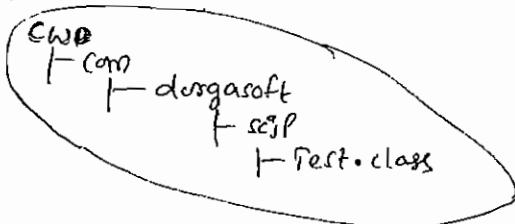
② javac -d .

destination to place  
Generated .class files



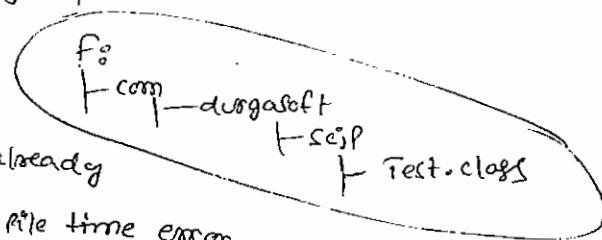
current working directory

Generated .class file will be placed in corresponding package structure.



- If the corresponding package structure not already available then this command by itself will create corresponding package structure.
- As destination instead of .(dot) we can take any valid directory name

Ex: `[javac -d F: Test.java]`



- If the specified directory not already available then we will get compile time error.

Ex:

`[javac -d Z: Test.java]`

If Z: not available then we will get compilation error saying directory not found: Z:

- At the time of execution we have to use fully qualified name

Ex:

`[java com.durgsoft.scip.Test]`

Op: Package demo.

### Conclusion 1

In any Java source file there can be atmost one package statement i.e more than one package statement is not allowed otherwise we will get compilation error.

Ex:

```
package Pack1;  
package Pack2;  
public class A  
{  
}
```

C:E: class, interface or enum expected.

### Conclusion 2

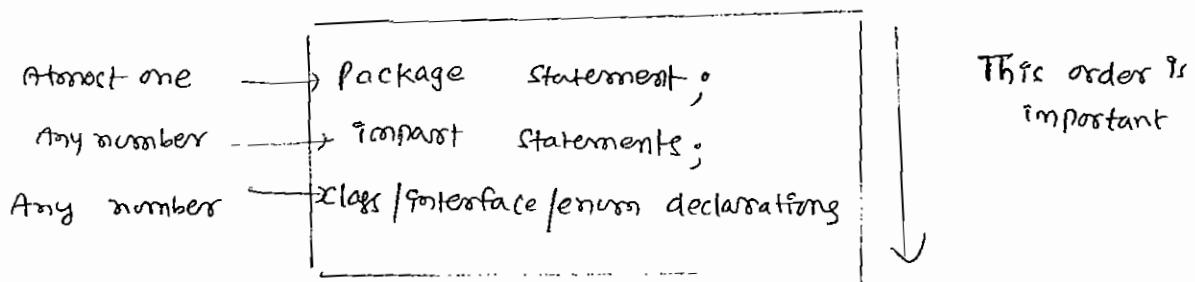
If any Java program the first non comment statement should be package statement (if it is available) otherwise we will get compilation error.

Ex:

```
import java.util.*;  
package Pack1;  
public class A  
{  
}
```

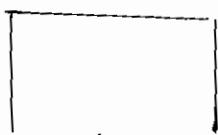
C:E: class, interface or enum expected.

The following is valid java source file structure.

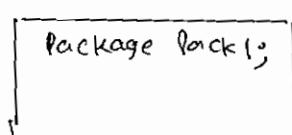


#### Notes

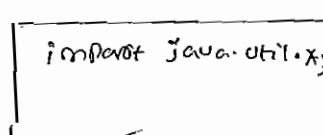
- An empty source file is a valid java program Hence the following are valid java source files



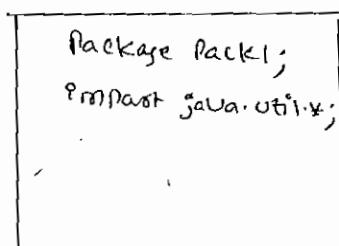
✓ Test.java



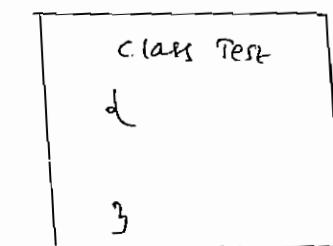
✓ Test.java



✓ Test.java



✓ Test.java



✓ Test.java

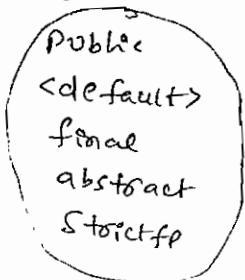
## ② CLASS LEVEL MODIFIERS

Whenever we are writing our own classes we have to provide some information about our class to the JVM like

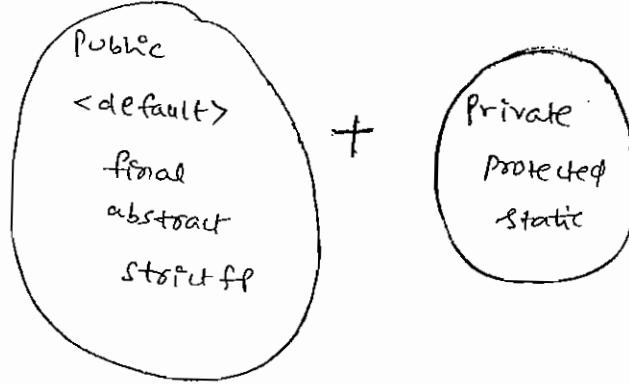
- ① whether this class can be accessible from anywhere or not
- ② whether child class creation is possible or not
- ③ whether object creation is possible or not etc...

We can specify this information by using appropriate modifier

- \* The only applicable modifiers for top level classes are



But for inner classes the applicable modifiers are



11. Private class  
12 }  
13 }  
14. static class  
} }  
15. `sum()`  
16. `set("text")`  
}

Ex:

private class Test

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

C.E: `modifiers private` not allowed here.

### Access Specifiers vs Access Modifiers

Public, Private, Protected, default are considered as Access specifiers except these remaining are considered as modifiers.

but this rule is applicable only for old languages like C++.

in Java all are considered as modifiers only there is no word like Specifier.

Ex:

private class Test

```
{  
    }  
}
```

C.E: `modifier private` not allowed here

### Public classes

Ex: If a class declared as `public` then we can access that class from anywhere

```
package pack1;  
public class A  
{  
    public void m1()  
    {  
        System.out.println("Hello");  
    }  
}
```

`javac -d . A.java`

pack1  
+ A.class

```
package Pack2;  
import pack1.A;  
class B  
{  
    public void main(String[] args)  
    {  
        A a = new A();  
        a.m1();  
    }  
}
```

`Java -d . B.java`

`java Pack2.B`

O/P: Hello

(111)

If class A is not public then while compiling B class we will get compilation error saying Pack1.A is not Public in Pack1; so cannot be accessed from out side package

### default classes

If a class declared as default then we can access that class only within the current package i.e. from outside package we can't access. Hence default access is also known as Package Level access

### final modifier

final is the modifier applicable for classes, method and variables

#### final method:

- whatever methods parent has by default available to the child through inheritance. If the child not satisfied with parent method implementation then child is allowed to redefine that method based on its requirement. This process is called overriding.
- If the parent class method is declared as final then we can't override that method in the child class because its implementation is final.

Ex:

```
class P
{
    public void property()
    {
        System.out.println("Cath + Land + Gold");
    }
}

class C extends P
{
    public void marry()
    {
        System.out.println("Subbalakshmi");
    }
}
```

C.e: marry() in C cannot override marry() in P; overridden method is final

#### final class:

Class is if a class declared as final we can't extend functionality of it possible for final classes. i.e. Inheritance is not

Ex: final class P

{

} class C extends P

{

}

C: can not inherit from final P

Note:



Every method present inside a final class is always final by default.  
But every variable present inside final class need not be final.

Ex:

final class P

{

    static int x=10;

    P p = new P();  
    p.x = 777;

    System.out.println(x);

    x = 777;  
    System.out.println(x);

}

}

Output: 10  
777

\* The main advantage of final keyword is we can achieve security and we can provide unique implementation but the main disadvantage of final keyword is we are missing key benefits of OOPS: inheritance (because of final classes) and polymorphism (because of final methods) hence if there is no specific requirement then it is not recommended to use final keyword.

## Abstract modifier

Abstract is the modifier applicable for classes and methods but not for variables

### Abstract method:

Even though we don't know about implementation still we can declare a method with abstract modifier i.e. for abstract methods only declaration is available but not implementation. Hence abstract method declaration should ends with ; (semicolon)

Ex:

public abstract void m1(); ✓

public abstract void m1(); } X

- child class is responsible to provide implementation for parent class abstract methods.

Ex: abstract class vehicle

```

    {
        abstract public int getNoOfWheels();
    }

    class Bus extends vehicle {
        public int getNoOfWheels() {
            return 7;
        }
    }
  
```

class Auto extends vehicle

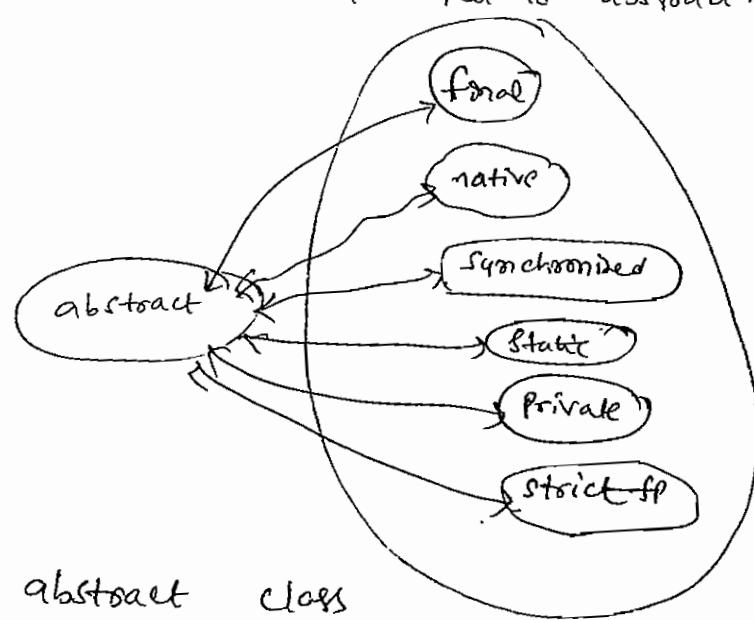
```

    {
        public int getNoOfWheels() {
            return 3;
        }
    }
  
```

\* By declaring abstract method in the parent class we can provide guidelines to the child classes such that which methods compulsory child has to implement.

\* abstract method never talks about implementation if any modifier talks about implementation then it forms illegal combination with abstract modifier

The following are various illegal combination of modifiers for methods with respect to abstract.



Ex:

abstract final void m();

GE: illegal combination of modifiers: abstract and final

for any java class if we ~~don't~~ are not allowed to create an object (because of partial implementation) such type of class we have to declare with abstract modifier

i.e for abstract classes instantiation is not possible.

Ex: abstract class Test

```

    {
        public void main(String[] args) {
            Test t = new Test();
        }
    }
  
```

GE: Test is abstract: can not be instantiated

## abstract class vs abstract method

- ① If a class contains atleast one abstract method then compulsory we should declare class as abstract. otherwise we will get compiletime error.

### Reason

If a class contains atleast one abstract method then implementation is not complete and hence it is not recommended to create object. To restrict object instantiation compulsory we should declare class as abstract.

\* ②

Even though class doesn't contain any abstract method still we can declare class as abstract if we don't want instantiation. i.e abstract class can contain zero number of abstract methods.

### Ex1

HttpServlet class is abstract but it doesn't contain any abstract methods.

### Ex2

Every Adapter class is recommended to declare as abstract but it doesn't contain any abstract methods.

bits

Ex1:

```
class P
{
    public void m1();
}
```

C.E:

missing method body, or declare abstract

(2)

```
class P
{
    public abstract void m1() { }
}
```

C.E: abstract methods can not have a body

#(3)

```
class P
{
    public abstract void m1();
}
```

C.E:

P is not abstract and does not override abstract method M1()

- (15)
- ③ If we are extending abstract class then for each & every abstract method of parent class we should provide implementation otherwise we have to declare child class of abstract.  
 → In this case next level child class is responsible to provide implementation

Ex:

abstract class P

```

  {
    public abstract void m1();
    public abstract void m2();
  }

  class C extends P
  {
    public void m1() { }
  }
  
```

C Err

C is not abstract and does not override abstract method m2() in P

## final vs abstract

- ① Abstract methods compulsory we should override in child classes to provide implementation. whereas as we can't override final methods. Hence final, abstract combination is illegal combination for methods.
- ② For final classes we can't create child class whereas for abstract classes we should create child class to provide implementation. Hence final, abstract combination is illegal for classes.
- \*③ Abstract class can contain final method whereas final class can't contain abstract method.

Ex:

```

abstract class Test
{
  public final void m1()
}
  
```

Note:

It is highly recommended to use abstract modifier because it promotes several OOP features like ~~ab~~ inheritance & Polymorphism

final class Test

```

{
  public abstract void m1();
}
  
```



## Strictfp modifier (Strict floating point)

- 1) Introduced in 1.2 version
- 2) we can ~~use~~ strictfp for classes and methods but not for variables
- 3) usually the result of floating point arithmetic is carried from platform to platform if we want platform independent results for floating point arithmetic then we should go for strictfp modifier

### Strictfp method

If a method declared as strictfp all floating point calculations in that method has to follow IEEE 754 standard so that we will get platform independent results.

- abstract modifier never talks about implementation. whereas strictfp always talks about implementation. Hence abstract strictfp combination is illegal for methods.

### Strictfp class

If a class declared as strictfp then every floating point calculation present in every concrete method has to follow IEEE 754 standard so that we will get platform independent results.

- \* We can declare abstract strictfp combination for classes i.e abstract strictfp combination is legal ~~for~~ for classes but illegal for methods

Exe

```
abstract strictfp class Test
{
    abstract strictfp void m();
}
```

illegal combination of modifiers:  
abstract & strictfp

### ③ Member Level Modifiers (Method (or) variable level modifiers)

#### ① Public members:

If a member declared as Public then we can access that Member from anywhere, but corresponding class should be visible i.e. before checking member visibility we have to check class visibility

Ex:-

```
package Pack1;  
class A  
{  
    public void m1() {  
        System.out.println("A class method");  
    }  
}
```

javac -d . A.java

```
package Pack2;  
import Pack1.A;  
class B  
{  
    public void main(String[] args)  
    {  
        A a = new A();  
        a.m1();  
    }  
}
```

javac -d . B.java

C-E:

Pack1.A is not Public in Pack1.  
Can not be accessed from outside package.

In the above example even though m1() method is Public we can't access from outside package because corresponding class A is not public, i.e. if both class & method are public then only we can access the method from outside package.

#### ② default members:

If a member declared as default then we can access that member only within the current package i.e. from outside of the package we can't access. Hence default access is also known as Package Level access.

#### ③ private members

If a member is private then we can access that member only within the class i.e. from outside of the class we can't access.

- abstract methods should be available to the child classes to provide implementation where as private methods are not available to the child classes. Hence private abstract combination is illegal for methods.

#### ④ Protected members (The most misunderstood modifier in java)

If a member declared as Protected then we can access that member anywhere within the current package but only in child classes of outside package.

**Protected = <default> + kfdg**

\* We can access protected members within the current package anywhere either by using parent reference or by using child reference.

but we can access protected members in outside package only in child classes and we should use child reference only i.e Parent reference can not be used to access protected members from outside package.

Ex:

```
Package Pack1;  
Public class A  
{  
    Protected void m1()  
    {  
        System.out.println("the most misunderstood  
        modifier");  
    }  
}  
  
class B extends A  
{  
    Public void main(String args)  
    {  
        ① A a = new A();  
        a.m1();  
        ② B b = new B();  
        b.m1();  
        ③ A q = new B();  
        q.m1();  
    }  
}
```

```
Package Pack2;  
Import Pack1.A;  
Class C extends A  
{  
    Public void main(String args)  
    {  
        ① A a = new A();  
        a.m1(); X  
        ② C c = new C();  
        c.m1();  
        ③ A q = new C();  
        q.m1(); X  
    }  
}
```

C.f: M1() has protected access in Pack1.A



- We can access protected members from outside package only for child classes and we should use that child class reference only.
- for example from D class if we want to access we should use D class reference only

Ex:

Package Pack2

```
import pack1.A;
class C extends A
{
}
class D extends C
{
    public static void main(String[] args)
}
```



- ① A a = new A();  
a.m1();
- ② C c = new C();  
c.m1();
- ③ D d = new D();  
d.m1();
- ④ A a1 = new C();  
a.m1();
- ⑤ A a1 = new D();  
a1.m1();
- ⑥ C c1 = new D();  
c1.m1();

Note: m1() has protected access in pack1.A

Summary table of private, <default>, protected & public modifiers

Visibility	Private	<default>	Protected	Public
with in the same class	✓	✓	✓	✓
from child class of same package	✗	✓	✓	✓
from non-child class of same package	✗	✓	✓	✓
from child class of outside package	✗	✗	✓ [we should use child reference only]	✓
from non-child class of outside package	✗	✗	✗	✗

- The most restricted access modifier is private.
- The most accessible modifier is public

private < default < protected < public

Recommended modifier for Data members (variable) is private  
 but recommended modifier for methods is public

29/09/2014

### final variables

- final instance variables.
- If the value of a variable is varied from Object to object such type of variables are called instance variables.
- for every object a separate copy of instance variables will be created.
- for instance variables we are not required to perform initialization explicitly Jvm will always provide default values.

Ex:

```

class Test
{
    int x;
    public void main(String[] args)
    {
        Test t=new Test();
        System.out.println(t.x); // 0
    }
}
```

- If the instance variable declared as final then compulsory we have to perform initialization explicitly whether we are using or not.

Ex: class Test

```

    {
        final int x;
    }
```

Ex: Variable x might not have been initialized

### Rules:

for final instance variable compulsorily we should perform initialization before constructor completion.

i.e. The following are various places for initialization

- at the time of declaration
- inside instance block
- inside constructor

① at the time of declaration

```
Class Test  
{  
    final int x=10;  
}
```

② Inside instance block

```
Class Test  
{  
    final int x;  
    {  
        x=10;  
    }  
}
```

③ Inside constructor

```
Class Test  
{  
    final int x;  
    Test()  
    {  
        x=10;  
    }  
}
```

These are the only possible places to perform initialization for final instance variables.

If we are trying to perform initialization anywhere else then we will get compiletime error.

Ex:

```
Class Test  
{  
    final int x;  
    public void m()  
    {  
        x=10;  
    }  
}
```

Can not assign a value to final variable x

② final static variable:

If the value of a variable is not carried from object to object such type of variables are not recommended to declare as instance variables. We have to declare those variables at class level by using static modifier.

In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.

\* for static variables it is not required to perform initialization explicitly JVM will always provide default values

Ex:

```
Class Test  
{  
    static int x;  
    public static void main(String[] args)  
    {  
        System.out.println(x);  
    }  
}
```

\* If the static variable declared as final then compulsorily we should perform initialization explicitly. otherwise we will get compiletime error and JVM won't provide any default values

Ex: Class Test

```
class Test {  
    final static int x;  
}
```

GE:

variable x might not have been initialized

Rules:

for final static variables compulsorily we should perform initialization before classloading completion. i.e. The following are various places for this

① At the time of declaration :

```
class Test {  
    final static int x>10;  
}
```

② Inside static Block

```
class Test {  
    static {  
        final static int x;  
        x=10;  
    }  
}
```

These are the only possible places to perform initialization for final static variables. if we are trying to perform initialization anywhere else then we will get compiletime errors

Ex: Class Test

```
class Test {  
    final static int x;  
    public void m1() {  
        x=10;  
    }  
}
```

GE:

Can not assign a value to final variable x

③ final Local Variables:

Some times to meet temporary requirements of the programmer we have to declare variables inside a method (or) block (or) constructor. such type of variables are called local variables (or) temporary variables (or) stack variables (or) automatic variables

for local variables JVM won't provide any default values ~~compile~~<sup>(93)</sup>.  
we should perform initialization explicitly before using that local variable. i.e if we are not using then it is not required to perform initialization for local variable.

Ex: class Test  
{  
 public void main(String[] args)  
 {  
 int x;  
 System.out.println("Hello");  
 }  
}

Output: Hello

class Test  
{  
 public void main(String[] args)  
 {  
 int x;  
 System.out.println(x);  
 }  
}

C.E: variable x might not have been initialized.

Even though local variable is final before using only we have to perform initialization i.e if we are not using then it is not required to perform initialization even though it is final.

Ex: class Test  
{  
 public void main(String[] args)  
 {  
 final int x;  
 System.out.println("Hello");  
 }  
}

Output: Hello

The only applicable modifier for local variable is final by mistake if we are trying to apply any other modifier then we will get compile time error.

Ex: class Test  
{  
 public void main(String[] args)  
 {  
 public int x=10;  
 private int x=10;  
 protected int x=10;  
 static int x=10;  
 transient int x=10;  
 volatile int x=10;  
 final int x=10;  
 }  
}

C.E: illegal start of expression

Note If we are not declaring any modifier then by default it is default but this rule is applicable only for instance & static variables but not for local variables.

### \* formal parameters

formal parameters of a method simply acts as local variables of that method hence formal parameter can be declared as final. If formal parameter declared as final ~~we~~ then within the method we can't perform reassignment.

Ex:

Class Test

{

    P S v main (String[] args)

{

        M1(10, 20); → actual Parameters

{

        P S v m1 (final int x, int y)

{

            if x = 100; → C.E → formal Parameters

            y = 200; → Can not assign a value to final variable x

            SOP ("x + " + " + y);

} }

### Static modifiers

Static is the modifier applicable for methods and variables but not for classes.

We can't declare top level class with static modifiers but we can declare inner class as static (such type of inner classes are called static nested classes).

In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.

Class Test

{ static int x = 10;

    int y = 20;

    P S v main (String[] args)

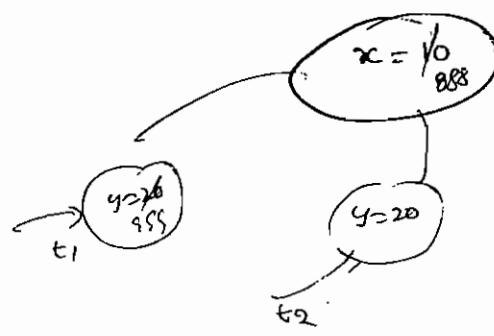
        Test t1 = new Test();

        t1.x = 888;

        t1.y = 999;

        Test t2 = new Test();

        SOP ("t1.x + " + " + t2.y);



- we can't access instance members directly from static area but (Q5)  
we can access from instance area directly. we can access static members from both instance and static area's directly.

Exe

Consider the following declarations

I. int x=10;  
II. static int x=10;  
III. public void m1()  
{  
    System.out.println("Hello");  
}  
IV. public static void m2()  
{  
    System.out.println("Hello");  
}

✓ I & III

✗ II I & IV → C.E

✗ III & IV

✗ IV & III

Non static variable  
x can not be  
referenced from  
static context

With in the same class which of the  
above declarations we can take  
simultaneously

(E) → class Test  
{  
    static int x=10;  
    int y=20;  
}

✗ (E) I & II → C.E  
✗ (F) III & IV  
    Variable x is already  
    defined in Test  
    C.E  
    (m1,2) is already  
    defined in Test

class Test  
{  
    public void m1()  
    {  
        System.out.println("Hello");  
    }  
    public static void m2()  
    {  
        System.out.println("Hello");  
    }  
}

Case 1

Overloading concept but JVM can always call static methods including main method only.

Exe

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

Op: String[]

other overloaded method we have to call just like a normal method call.

## Case 2

Inheritance concept applicable for static methods including main method. Hence while executing child class if child doesn't contain main method then parent class main method will be executed.

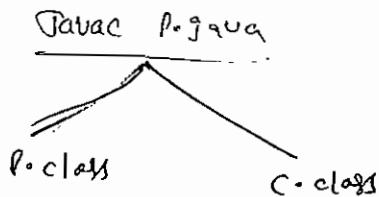
Ex:

```

class P {
    public static void main (String args) {
        System.out.println ("Parent main");
    }
}

class C extends P {
}

```



Java P

O/P: Parent main

Java C

O/P: Parent main

## Case 3

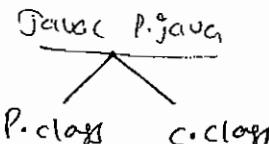
```

class P {
    public static void main (String args) {
        System.out.println ("Parent main");
    }
}

class C extends P {
    public static void main (String args) {
        System.out.println ("Child main");
    }
}

```

It is  
method  
hiding  
but not  
overriding



Java P

O/P: Parent main

Java C

O/P: child main

- \* At learning overriding Overriding concept applicable for static methods but it is not applicable if it is method hiding.
- \* For static methods overloading & inheritance concepts are applicable but overriding concept is not applicable. but instead of overriding method hiding concept is applicable.

- Inside method implementation if we are using atleast one instance variable then that method talks about a particular object, hence we should declare method as instance method.
- Inside method implementation if we are not using any instance variable then this method no way related to a particular object, hence we have to declare such type of method as static method irrespective of whether we are using static variables or not.

Ex:

```
class Student
{
    String name;
    int rollno;
    int marks;
    static String cname;
}
```

```
instance method { getStudentInfo()
{
    return name + " " + marks;
}
getCollegeInfo()
{
    return cname;
}}
```

```
Static method { getAverage (int x, int y)
{
    return (x+y)/2;
}
getCompleteInfo()
{
    return name + " " + rollno + " " + marks + " " + cname;
}}
```

- for static methods implementation should be available whereas for abstract methods implementation is not available hence abstract static combination is illegal for methods.

## Synchronized modifier

Synchronized is the modifier applicable for methods and blocks but not for classes and variables.  
 If multiple threads trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem this is called 'race condition'. We can overcome this problem by using synchronized keyword.

If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.  
 But the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems. Hence if there is no specific requirement then it is not recommended to use synchronized keyword.

Synchronized method should compulsorily contain implementation whereas abstract method doesn't contain any implementation hence abstract synchronized is illegal combination of modifiers for methods.

### Native modifiers

Native is the modifier applicable only for methods and we can't apply anywhere else.

The methods which are implemented in non-Java (mostly C or C++) are called native methods (or) foreign methods.

The main objectives of native keyword are

- (1) To improve performance of the system
- (2) To achieve machine level or memory level communication
- (3) To use already existing legacy non-Java code.

### PseudoCode to use native keyword in Java

① Load native library

```
class Native
{
    static
    {
        System.loadLibrary("native library path");
    }
}
```

② declare a native method

```
public native void m1();
```

③ invoke a native method

```
Native n = new Native();
n.m1();
```

For native methods implementation is already available in old languages like C/C++ and we are not responsible to provide implementation hence native method declaration should end with ; (semicolon)

Ex:

```
(1) public native void m1();
```

X (2) public native void m1 { }

∴ native methods can not have a body

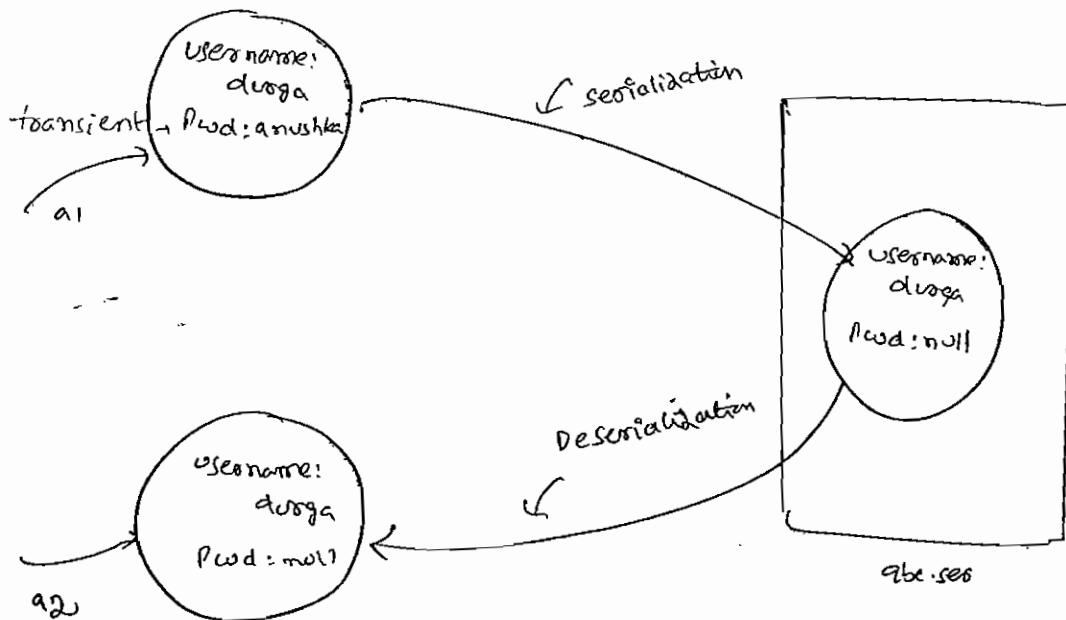
- for native methods implementation is already available in old languages but for abstract methods implementation should not be available hence we can't declare native method as abstract i.e. native, abstract combination is illegal combination for methods.
- we can't declare native method as `strictfp` because there is no guarantee that old languages follow IEEE 754 standard. Hence native strictfp combination is illegal combination for methods.
- The main advantage of native keyword is performance will be improved but the main disadvantage of native keyword is it breaks platform independent nature of java.

### transient keyword

Transient is the modifier applicable only for variables.

We can use transient keyword in serialization context.

- at the time of serialization if we don't want to save the value of a particular variable to meet security constraint then we should declare that variable as transient
  - at the time of serialization JVM ignores original value of transient variables and save default value to the file. Hence
- transient means not to serialize



## Volatile

Volatile is the modifier applicable only for variables and we can't apply anywhere else.

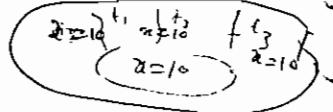
If the value of a variable keep on changing by multiple threads then there may be a chance of data inconsistency problem we can solve this problem by using volatile modifier.

If a variable declared as volatile then for every thread JVM will create a separate local copy.

So that there is no effect on the remaining threads.

The main advantage of volatile keyword is we can overcome data inconsistency problem but the main disadvantage of volatile keyword is creating and maintaining a separate copy for every thread increases complexity of programming and creates performance problems. Hence if there is no specific requirement it is never recommended to use volatile keyword and it is almost deprecated keyword.

Volatile means the value never changes whereas volatile final is illegal combination for variable.



modifier	closes			variables	blocks	interface		enum		constants
	outer	inner	methods			outer	inner	outer	inner	
public	✓	✓	✓	✓	X	✓	✓	✓	✓	✓
private	X	✓	✓	✓	X	X	X	✓	✓	✓
protected	X	✓	✓	✓	X	X	X	✓	✓	✓
<default>	✓	✓	✓	✓	X	✓	✓	✓	✓	✓
final	✓	✓	✓	✓	X	X	X	✓	✓	✓
abstract	✓	✓	✓	X	X	✓	✓	X	X	X
static	X	✓	✓	✓	X	X	X	X	X	X
synchronized	X	X	✓	X	✓	X	✓	X	✓	X
native	X	X	✓	X	✓	X	X	X	X	X
strictfp	✓	✓	✓	X	X	✓	✓	✓	✓	X
transient	X	X	X	✓	X	X	X	X	X	X
volatile	X	X	X	✓	X	X	X	X	X	X

- The only applicable modifier for local variable is final
- The only applicable modifier for constructors are public, private, protected, default
- The modifiers which are applicable only for methods native
- The modifiers which are applicable only for variables volatile & transient
- The modifiers which are applicable for classes but not for interface final
- The modifiers which are applicable for classes but not for enum final and abstract.

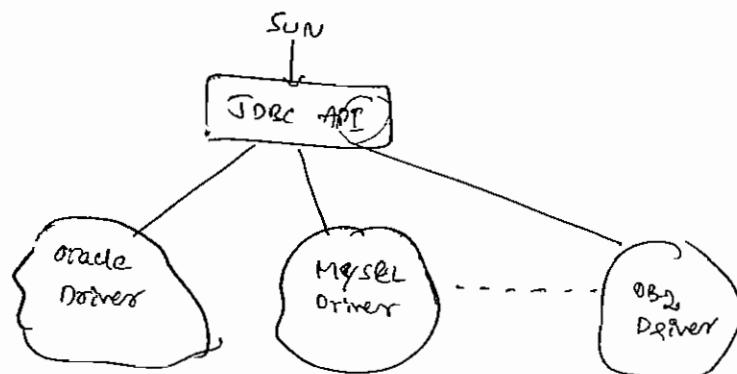
## Interfaces

- (1) Introduction
- (2) Interface Declaration & Implementation
- (3) extends vs implements
- (4) Interface methods
- (5) Interface Variables
- (6) Interface naming conflicts
  - (i) method naming conflicts
  - (ii) variable naming conflicts
- \* (7) marker interface
- (8) Adapter classes
- (9) Interface vs abstract class vs concrete class
- \* (10) Differences b/w interfaces & abstract class
- (11) various important conclusions

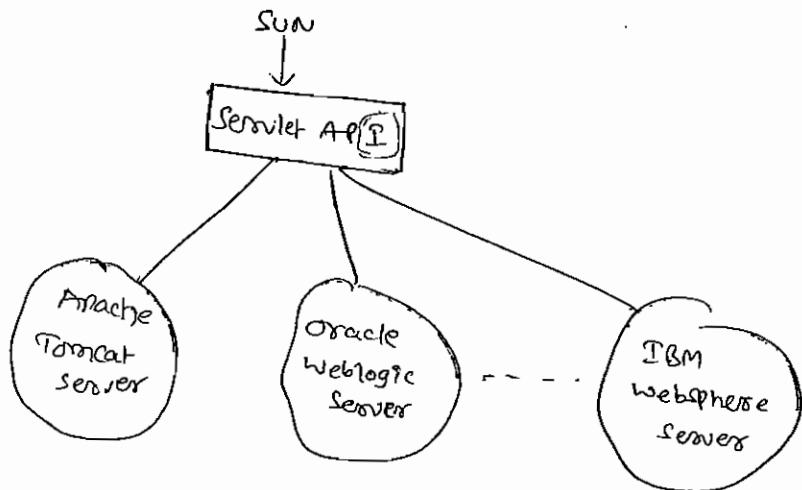
### (1) Introduction

Defn: Any service requirement specification (SRS) is considered as an interface.

Ex: JDBC API acts as requirement specification to develop Database driver. Database vendor is responsible to implement this JDBC API.



Ex 2 e Servlet API acts as requirement specification to develop webserver. Webserver vendor is responsible to develop implementation of Servlet API.

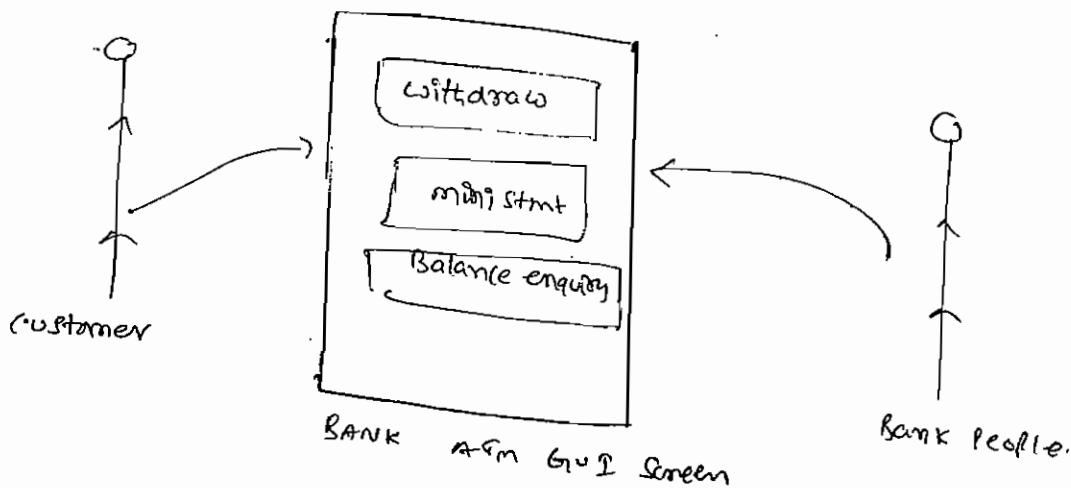


Def 2 e

from client point of view an interface defines the set of services what he is expecting.  
from service provider point of view an interface defines the set of services what he is offering.  
Hence any contract between client and service provider is considered as an interface.

Ski

Through bank ATM GUI screen bank people are highlighting the set of services what they are offering. at the same time the same GUI screen represents the set of services what customer is expecting. Hence this GUI screen acts as contract between customer and bank people.



Def 3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

### Summary Definition

Any service requirement specification (or) any contract between client and service provider (or). 100% pure abstract class is nothing but interface.

### (2) Interface Declaration & Implementation

- (1) Whenever we are implementing an interface for each and every method of that interface we have to provide implementation otherwise we have to declare class as ~~pure~~ abstract. Then next level child class is responsible to provide implementation.
- (2) Every interface method is always public and abstract whether we are declaring or not. Hence whenever we are implementing an interface method compulsory we should declare as public otherwise we will get compilation error.

Ex:

```
interface Interf
{
    void m1();
    void m2();
}

class ServiceProvider implements Interf
{
    public void m1()
    {
    }

    class SubServiceProvider extends ServiceProvider
    {
        public void m2()
        {
        }
    }
}
```

### (3) Extends vs Implements

- (1) A class can extend only one class at a time
- (2) An interface can extend any number of interfaces simultaneously

ex: Interface A      Interface B

```
{
}
}

Interface C extends A, B
```

✓

- (3) A class can implement any number of interfaces simultaneously
- (4) a class can extend another class and can implement any number of interfaces simultaneously

Ex:

Class A extends B implements C, D, E

{

}

b18

(5) Which of the following is valid?

(1) A class can extend any number of classes at a time X

(2) A class can implement only one interface at a time X

(3) An interface can extend only one interface at a time X

(4) An interface can implement any number of interfaces simultaneously X

(5) a class can extend another class or can implement an interface but not both simultaneously X

(6) None of the above X

(7) Consider the following expression  
the following possibilities of X and Y for which of the above expression is valid.

(1) Both X and Y should be classes

(2) Both X and Y should be interfaces

(3) Both X and Y can be either classes or interfaces

(4) No restrictions

(2) X extends Y, Z

X, Y, Z should be interfaces

(3) X implements Y, Z

X → class  
Y, Z → interfaces

(4) X ~~ext~~ extends Y implements Z

X, Y → classes

Z → interface

(5) X implements Y extends Z

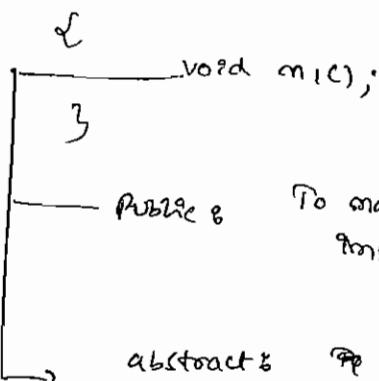
because we have to take  
extends first followed by  
interface

#### (4) Interface Methods

155

Every method present inside interface is always public and abstract whether we are declaring or not.

Ex:- Interface Interf



To make this method available to every implementation class

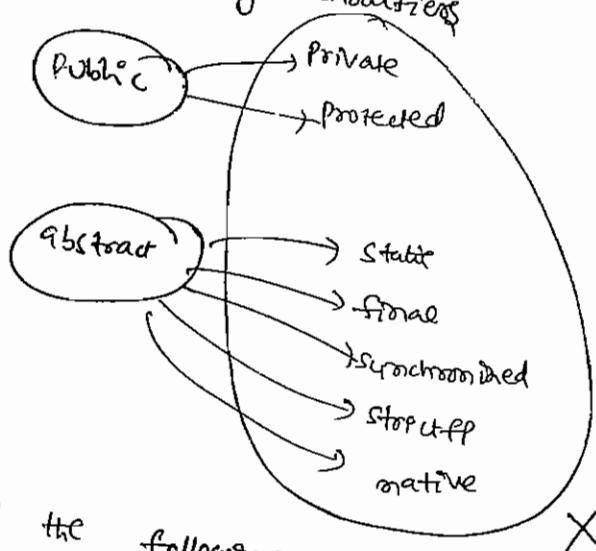
abstracts The implementation class is responsible to provide implementation

Hence inside interface the following method declarations are equal

void m1();  
Public void m1();  
Abstract void m1();  
Public abstract void m1();

As every interface with the method is always public & abstract we can't declare interface

modifiers



bit which of the following method declarations are allowed inside interface.

- X (1) public void m1() {}  
(2) private void m1();  
(3) protected void m1();  
(4) static void m1();  
(5) public abstract native void m1();  
(6) abstract public void m1();

## 5) Interface Variables

An interface can contain variables. The main purpose of interface variable is to define requirement level constants.

Every interface variable is always public static final whether we are declaring or not.

Interface Interf

{

int x=10;

}

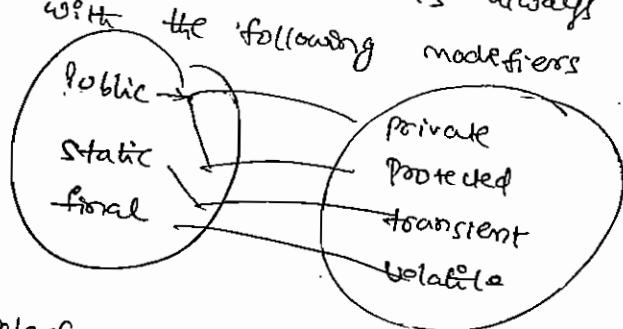
- public: To make this variable available to every implementation class.
- static: without existing object also implementation class has to access this variable.
- final:
  - If one implementation class changes value then remaining implementation classes will be effected to restrict this.
  - every interface variable is always final.

Hence with in the interface the following variable declarations are equal.

equal

int x=10;  
public int x=10;  
static int x=10;  
final int x=10;  
public static int x=10;  
public final int x=10;  
static final int x=10;  
public static final int x=10;

As every interface declare with the following variable is always modifiers



public static final we can't

transient → Interface no object  
↳ serializable required obj  
volatile → not a final

for interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compiletime error.

Ex: Interface Interf

{  
    int x;  
}

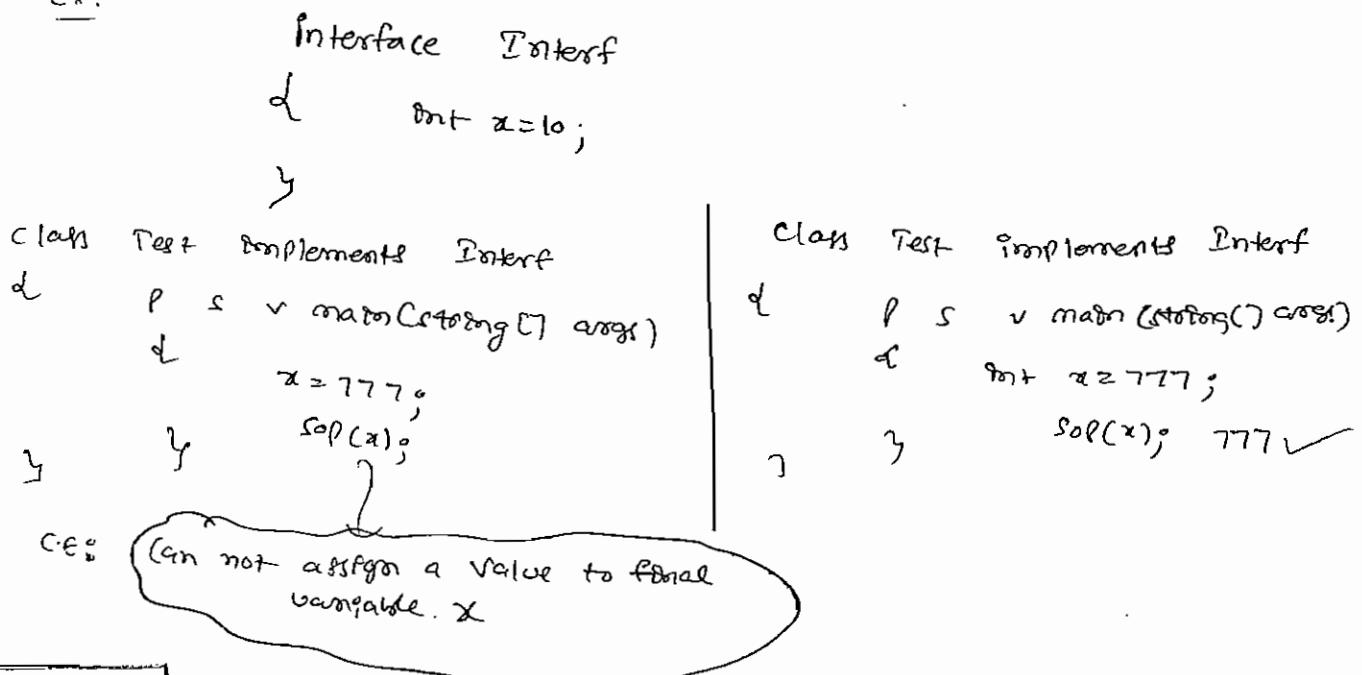
C.E.S = expected

Inside interface which of the following variable declarations are allowed. (Q1)

- (1) int x;
- (2) private int x=10;
- (3) protected int x=10; } X
- (4) volatile int x=10; }
- (5) transient int x=10; }
- (6) public static int x=10; ✓

- Inside implementation class we can access interface variables but we can't modify values

Ex:

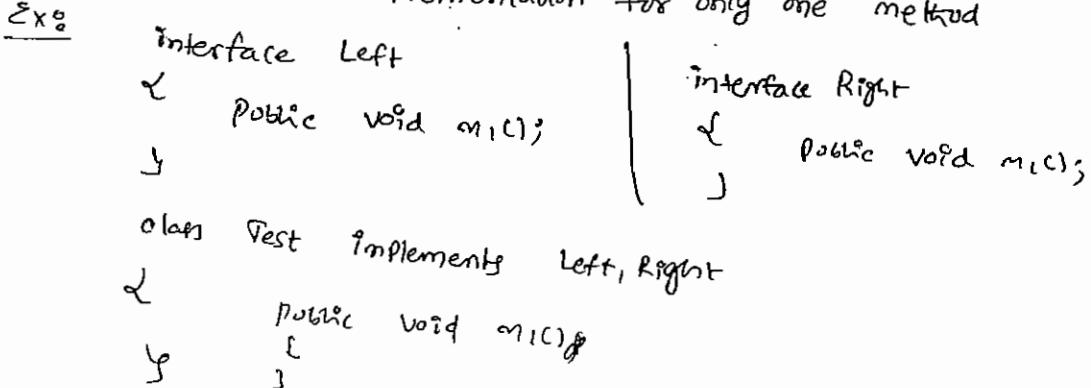


01/16/2014

## ⑥ Interface naming Conflicts

### ① method naming conflicts:

Case 1: if two interfaces contains a method with same signature and same return type then in the implementation class we have to provide implementation for only one method



Case 2 If two interfaces contain a method with same name but different argument types then in the implementation class we have to provide implementation for both methods and these methods acts as overloaded methods.

Ex 5

Interface Left

```
↳ Public void m1();  
↳
```

Interface Right

```
↳  
↳ Public void m1(int i);  
↳
```

Class Test implements Left, Right

```
{  
    ↳ Public void m1()  
    ↳  
    ↳ Public void m1(int i)  
    ↳  
    ↳ }  
overloaded  
methods
```

Case 3

If two interfaces containing a method with same signature but different return types then it is impossible to implement both interfaces simultaneously (if return types are not co-variant)

Ex 6

Interface Left

```
↳ Public void m1();  
↳
```

Interface Right

```
↳ Public int m1();  
↳
```

We can't write any Java class which implements both interfaces simultaneously

- (Q) Is a Java class can implement any number of interfaces simultaneously?  
A) Yes, except a particular case  
If two interfaces containing a method with same signature but different return types then it is impossible to implement both interfaces simultaneously.

④

Variable naming conflicts :

Two interfaces can contain a variable with the same name and there may be a chance of variable naming conflicts but we can solve this problem by using interface names.

Ex:

```

interface Left {
    int x=777;
}

interface Right {
    int x=888;
}

class Test implements Left, Right {
    public static void main(String[] args) {
        System.out.println(x); // Reference to x is ambiguous
        System.out.println(Left.x); 777
        System.out.println(Right.x); 888
    }
}

```

C.E: Reference to x is ambiguous



## Marker Interface

If an interface doesn't contain any methods and by implementing that interface if our objects will get some ability such type of interfaces are called marker interfaces (or) ability interface (or) Tag interface.

Ex: { Serializable(I), Cloneable(I), RandomAccess(I),  
 ↴ { SingleThreadModel(I), etc.....  
 These are marked for some ability

- Ex:
- 1) By implementing Serializable(I) interface our objects can be saved to the file & can travel across the network.
  - 2) By implementing Cloneable(I) our objects are in a position to produce exactly duplicate cloned objects.
- \* Q) Without having any methods how the objects will get some ability internally JVM is responsible to provide required ability.
- a) why JVM is providing required ability in marker interfaces?  
 To reduce complexity of programming and to make Java language as simple.
- Q) Is it possible to create our own marker Interface?  
 Yes, But customization of JVM is required.

## 8) Adapter classes

Adapter class is a simple java class that implements an interface with only empty implementation.

```
Interface X
{
    m1();
    m2();
    m3();
    .
    .
    !
    m1000();
}
```

```
Abstract class AdapterX implements X
{
    m1() { }
    m2() { }
    m3() { }
    m4() { }
    m5() { }
    .
    .
    .
    m1000() { }
```

If we implement an interface for each and every method of that interface compulsorily we should provide implementation whether it is required or not required.

```
Class Test implements X
{
```

```
    m3()
    {
        .
        .
        .
        m1() { }
        m2() { }
        m4() { }
        .
        .
        .
        m1000() { }
    }
```

}      *≡ 10 lines*

}      *999 lines of code.*

The problem in this approach is it increases length of the code and reduces readability.

We can solve this problem by using Adapter classes.

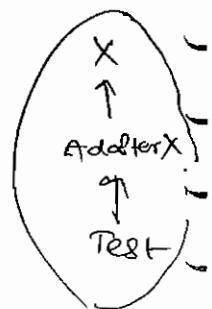
Instead of implementing interface if we extend Adapter class we have to provide implementation only for required methods and we are not responsible to provide implementation for each & every method of the interface.

Ex: So that length of the code will be reduced.

```
Class Test extends AdapterX
{
    m3()
    {
        .
        .
        .
    }
}

Class Sample extends AdapterX
{
    m7()
    {
        .
        .
        .
    }
}
```

```
Class Demo extends AdapterX
{
    m1000()
    {
        .
        .
        .
    }
}
```



Ex. We can develop a servlet in the following ③ ways

- ① By implementing `Servlet(I)` interface
- ② By extending `GenericServlet(AC)`
- ③ By extending `HttpServlet(AC)`

If we implement `Servlet` interface for each & every method of that interface we should provide implementation. It increases length of the code & reduces readability.

Instead of implementing `Servlet(I)` interface directly if we extend `GenericServlet` we have to provide implementation only for `service` method. and for all remaining methods we are not required to provide implementation. Hence more or less `GenericServlet` acts as Adapter class for `Servlet(I)` interface.

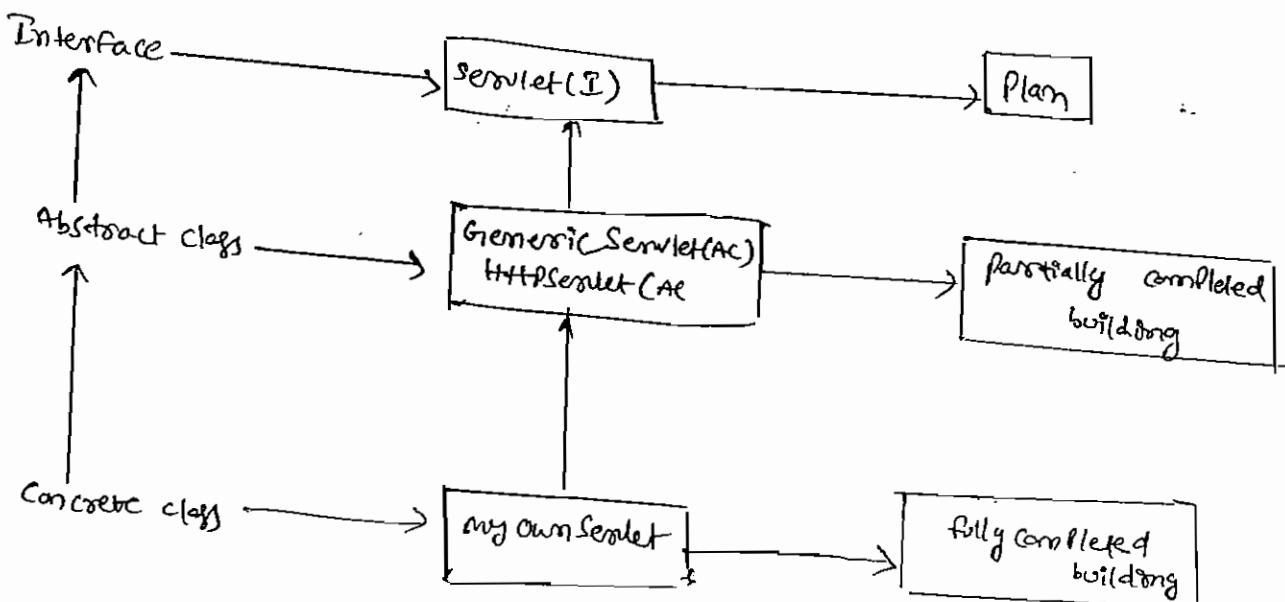
#### Notes

Marker interface and Adapter classes simplifies complexity of programming and these are best utilities to the programmer and programmers life will become simple.

⑨

#### Interface vs Abstract class vs Concrete class

- ① If we don't know anything about implementation just we have requirement specification then we should go for interface.  
Ex: `Servlet(I)`
- ② If we are talking about implementation but not completely (Partial implementation) then we should go for abstract class  
Eg: `GenericServlet(AC)`, `HttpServlet(AC)` etc...
- ③ If we are talking about implementation completely and ready to provide service then we should go for concrete class  
Eg: `MyOwnServlet`.



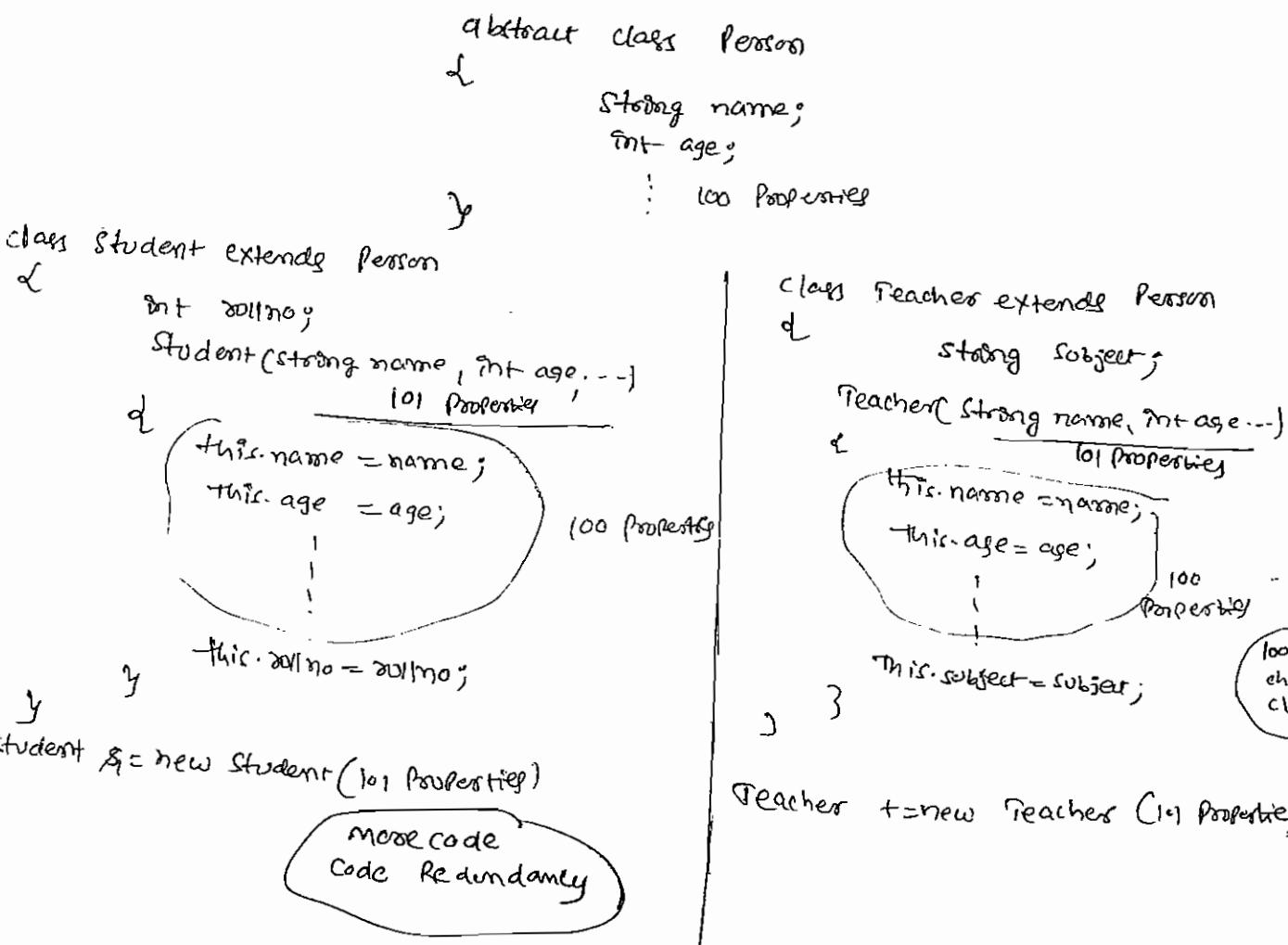
## \* 10 Differences b/w interface and Abstract class

Interface	Abstract class
① If we don't know anything about implementation and just we have requirement specification then we should go for interface.	① If we are talking about implementation but not completely (Partial Implementation) then we should go for Abstract class.
② Inside interface every method is always public and abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.	② Every method present inside Abstract class need not be public and abstract and we can take concrete methods also.
③ As every interface method is always public and abstract and hence we can't declare with the following modifiers private, protected, final, static, synchronized, native and strictfp	③ There are no restrictions on Abstract class method modifiers
④ Every variable present inside interface is always public static final whether we are declaring or not.	④ Every variable present inside Abstract class need not be public static final.
⑤ As every interface is always public static final we can't declare with the following modifiers private, protected, volatile & transient	⑤ There are no restrictions on Abstract class variable modifiers
⑥ For interface variables compulsorily we should perform initialization at the time of declaration only otherwise we will get compilation error.	⑥ For Abstract class variables we are not required to perform initialization at the time of declaration.
⑦ Inside interface we can <sup>not</sup> declare static and instance blocks	⑦ Inside Abstract class we can declare static & instance blocks
⑧ Inside interface we can not declare constructors	⑧ Inside Abstract class we can declare constructors

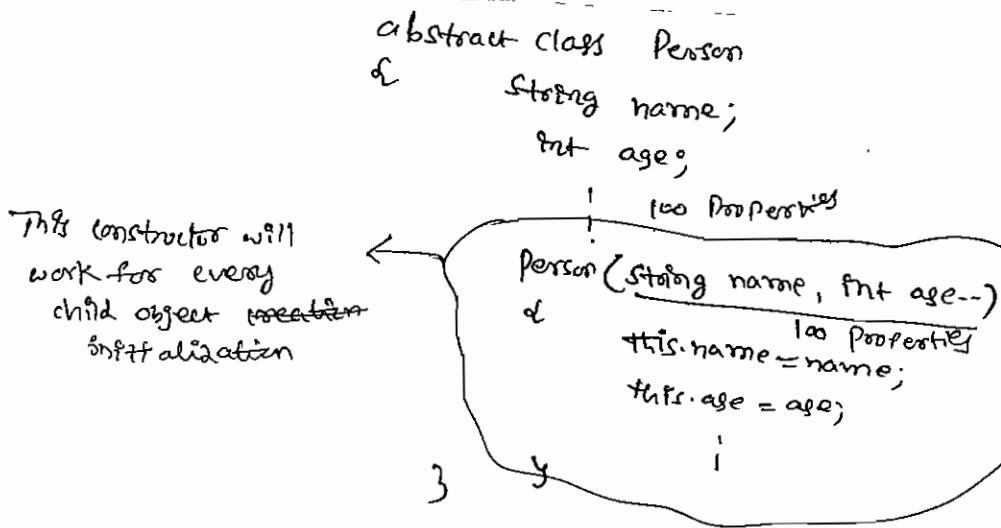
- Q) Anyway we can't create object for abstract class but abstract class can contain constructor what is the need?  
Abstract class constructor will be executed whenever we are creating child class object to perform initialization of child class object.

Approach 1

without having constructor in Abstract class

Approach 2:

with constructor inside Abstract class



```

class Student extends Person
{
    int rollno;
    Student(String name, int age)
    {
        super(100 Properties);
        this.rollno = rollno;
    }
}

```

```

Student s = new Student("101 Properties");

```

```

class Teacher extends Person
{
    String subject;
}
```

```

Teacher (String name, int age)
{
    101 Properties
}
```

```

sellers(100 Properties);
this.subject = subject;
}
```

```

Teacher t = new Teacher("101 Properties");

```

1000  
child  
classes

Copy code &  
Code Reusability

NOTE ①

Either directly or indirectly we can't create object for abstract class

- (o) any way we can't create objects for abstract class and interface but abstract class can contain constructors but interface doesn't contain constructors what is the reason? The main purpose of constructor is to perform initialization for the instance variables
- abstract class can contain instance variables which are required for child object. to perform initialization of those instance variables constructor is required for abstract class.
  - but every variable present inside interface is always public static final whether we are declaring or not and there is no chance of existing instance variable inside interface hence constructor concept is not required for interface.

~~when ever we are creating child class object parent object won't be created just parent class constructor will be executed for the child object~~

```

Ex:
class P
{
    P()
    {
        System.out.println("P");
    }
}

class C extends P
{
    C()
    {
        System.out.println("C");
    }
}

class Test
{
    public static void main(String[] args)
    {
        C c = new C();
        System.out.println(c.hashCode());
    }
}

```

- \* Q) Inside interface every method is always abstract and we can take only abstract methods in abstract class also then what is the difference b/w interface & abstract class i.e. is it possible to replace interface with abstract class?  
we can replace interface with abstract class but it is not a good programming practice  
This is something like recruiting IAS officer for screening activity  
→ If everything is abstract then it is highly recommended to go for interface but not for abstract class

### Approach 1

```
Abstract Class X
{
}
Class Test extends X
{
    ...
}
```

while extending Abstract class  
it is not possible to extend  
any other class & hence  
we are missing inheritance benefit

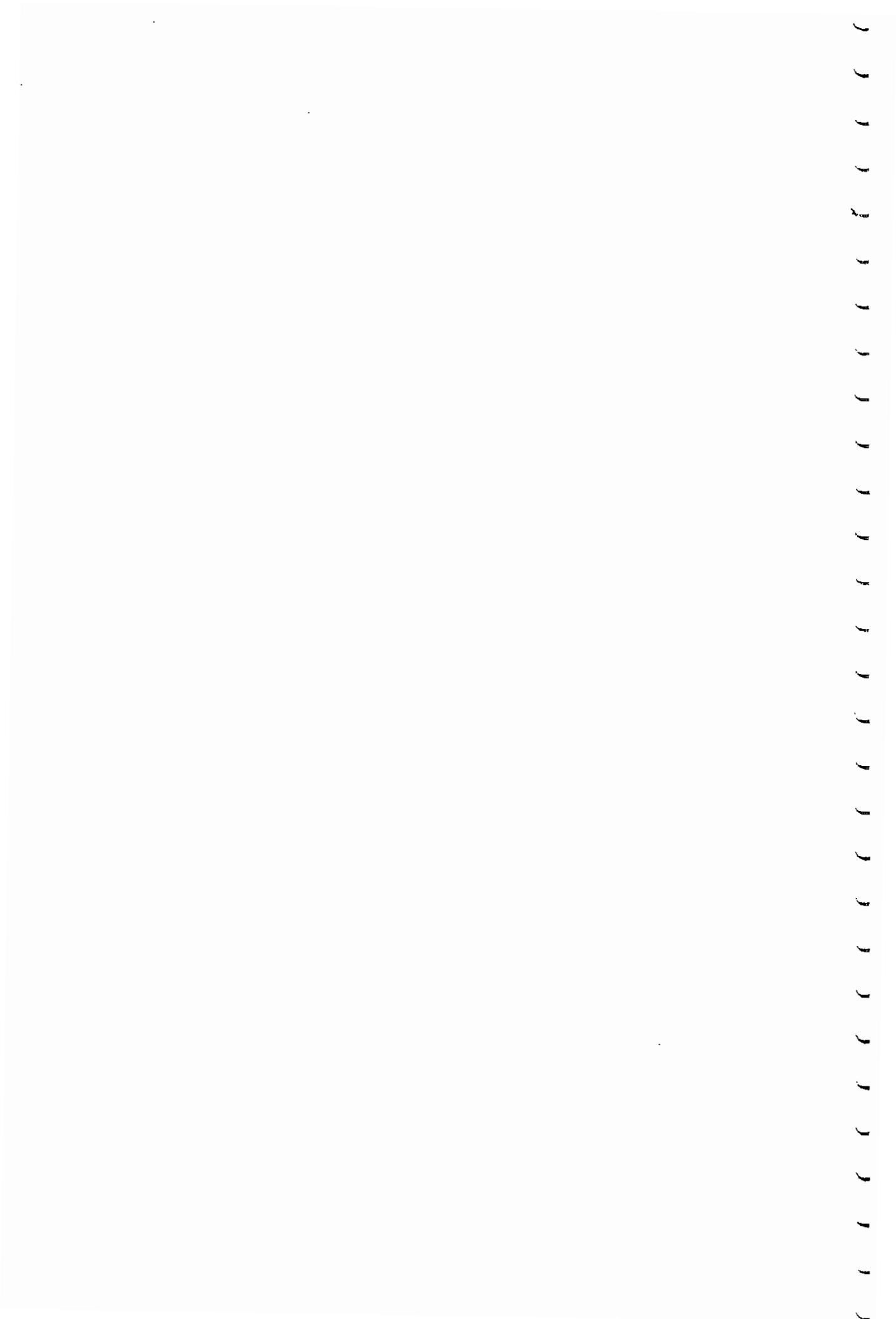
- ② Test t = new Test();  
⇒ This case object creation is costly  
Eg: Test t = new Test();  
(d mang)

### Approach 2

```
Interface X
{
}
Class Test implements X
{
    ...
}
```

while implementing interface we can extend  
some other class and hence we won't miss  
any inheritance benefit

- ② In this case object creation is not costly  
Eg: Test t = new Test();  
2sec



11-08-2014

# Exception Handling

14+

- (1) Introduction
- (2) Runtime Stack mechanism
- (3) Default Exception handling in Java
- (4) Exception Hierarchy
- (5) Customized Exception handling by using try-catch
- (6) Control flow in try-catch
- (7) Methods to print Exception information
- (8) try with multiple catch blocks
- (9) finally block
- (10) final vs finally vs finalize()
- (11) Control flow in try-catch-finally
- (12) Control flow in nested try-catch-finally
- (13) Various possible combinations of try-catch-finally
- (14) throw keyword
- (15) throws keyword
- (16) Various Possible compiletime errors in Exception Handling
- (17) Exception Handling keywords Summary
- (18) Customized Exceptions
- (19) Top-10 Exceptions
- (20) 1.7 Version Enhancements in Exception Handling
  - 1) try with resources
  - 2) Multi-Catch block

## Introduction

- An unwanted unexpected event that disturbs normal flow of the program is called Exception.
  - Eg: Sleeping Exception
  - TyrePuncturedException
  - PowercutException
- It is highly recommended to handle exceptions.
- Exception handling doesn't mean repairing an exception.  
It means providing alternative way to continue rest of the program normally.  
For example if our programming requirement is to read data from a file which is available at remote machine and at runtime if that remote machine file is not available then our program should not be terminated abnormally.

→ as alternative provide a local file to continue rest of the program normally.  
This way of providing alternative way is nothing but ExceptionHandling.

→ The main objective of Exception handling is graceful termination of the program  
(Normal termination of the program)

## ② Runtime Stack mechanism

- for every thread JVM creates a runtime stack
- All the method calls performed by that particular thread will be maintained in the Runtime stack
- each entry in the Runtime stack is called Stackframe (or) activation Record.
- Once if the method execution completes corresponding entry will be removed from the Runtime stack.
- Once all the methods execution completed then empty Runtime stack will be destroyed by JVM just before terminating the thread.
- This entire process is called Runtime stack mechanism

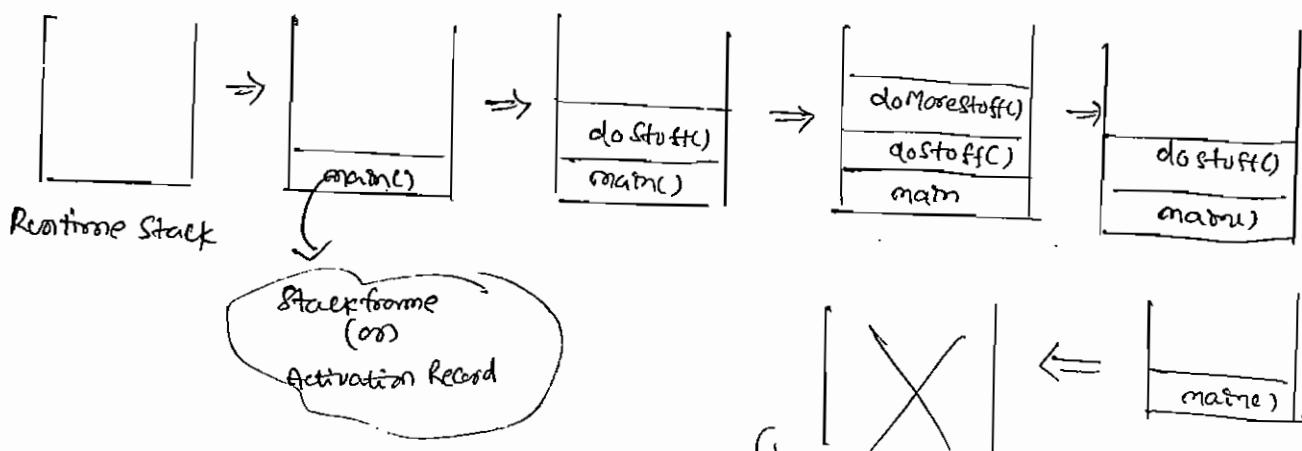
Ex:

Class Test {

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

public static void doStuff()
{
    doMoreStuff();
}

public static void doMoreStuff()
{
    System.out.println("Hello"); // Output - Hello
}
```



Empty Runtime stack will be destroyed by JVM just before terminating the thread.

### ③ Default Exception Handling in Java

In our program while executing any method if an exception is raised then the corresponding method is responsible to create exception object by including the following information.

- 1) Name of the exception
- 2) Description of the exception
- 3) StackTrace

- After creating Exception object method handovers that created exception object to the JVM
- JVM checks whether the corresponding method contains Exception handling code or not.
- If the method contains Exception handling code then it will be executed otherwise JVM terminates that method abnormally and removes corresponding entry from the runtime stack.
- JVM identifies the caller method and checks whether the caller method contains any Exception handling code or not
- If the caller method contains exception handling code then it will be executed otherwise JVM terminates caller method also abnormally and removes corresponding entry from the runtime stack.
- This process will be continued until main() method and if main() method also doesn't contain Exception handling code then JVM terminates main() method also abnormally and removes corresponding entry from the runtime stack.
- Then JVM handovers created exception object to the default exception handler, which is a component of JVM.
- ↳ DefaultExceptionhandler just prints exception information to the console in the following format and terminates the program abnormally

Exception in thread "thread-name" Name of the Exception : Description of the exception

#### StackTrace

```

Class Test
{
    public static void main(String[] args)
    {
        foo();
    }

    public static void foo()
    {
        bar();
    }

    public static void bar()
    {
        System.out.println("Hello");
    }
}

```

Name: **Test**  
 Description: (by zero)

Stack Trace: **bar()**, **foo()**, **main()**

bar()
foo()
main()

Runtime Stack

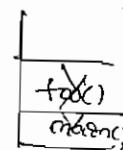
Exception in thread "main" java.lang.ArithmaticException : / by zero

at Test.bar()  
at Test.foo()  
at Test.main()

Ex 28

Class Test

```
g in DEX  
JVM  
name  
byz0  
at foo()  
main()  
public static void main(String[] args)  
{  
    foo();  
}  
public static void foo()  
{  
    bar();  
    System.out.println("Hello");  
}  
public static void bar()  
{  
    System.out.println("Hello"); // bar  
}
```



Runtime Stack

Output:  
Hello  
Exception in thread "main" java.lang.ArithmaticException : / by zero  
at Test.foo()  
at Test.main()

Ex 29

```
Class Test {  
    public static void main(String[] args)  
    {  
        foo();  
        System.out.println("Hello");  
    }  
    static void foo()  
    {  
        bar();  
        System.out.println("Hi");  
    }  
    public static void bar()  
    {  
        System.out.println("Hello");  
    }  
}
```

Output: Hello  
Hi

Exception in thread "main" java.lang.ArithmaticException : / by zero

at Test.main()

- If our program if every method terminated normally then only the program termination is normal termination
- If atleast one method terminated abnormally then the program termination is abnormal termination.

## ~ (4) Exception Hierarchy

(17)

- Throwable class acts as base class for Exception hierarchy i.e it is the root for all Exceptions and Errors
- Throwable class defines ② child classes
  - 1) Exception
  - 2) Error

### ① Exception

- most of the cases Exceptions will be raised because of our program logic and these are recoverable.

Ex: FileNotFoundException

- If our programming requirement is to read data from a remote machine file and at runtime if that file is not available we will get FileNotFoundException.
- but we can handle this Exception by providing a local file to continue rest of the program normally.

### ② Errors

Most of the cases errors won't be caused because of our program logic and these are due to lack of system resources.

- Errors are not recoverable

Ex: In our program if OutOfMemoryError occurs before a programme we can't do anything. System administrator (or) server administrator is responsible to increase HeapSize

Exception like "main"; java.lang.OutOfMemoryError

int[] x = new int[  
[21473648]

{  
    for(x);  
    // no exception occurred at  
    // compilation time  
    // exception occurred  
    // at runtime  
    // tracing ↗

### ③ Checked vs Unchecked Exceptions

#### Checked exception

The Exceptions which are checked by the compiler for smooth execution of the program at runtime are called Checked Exceptions

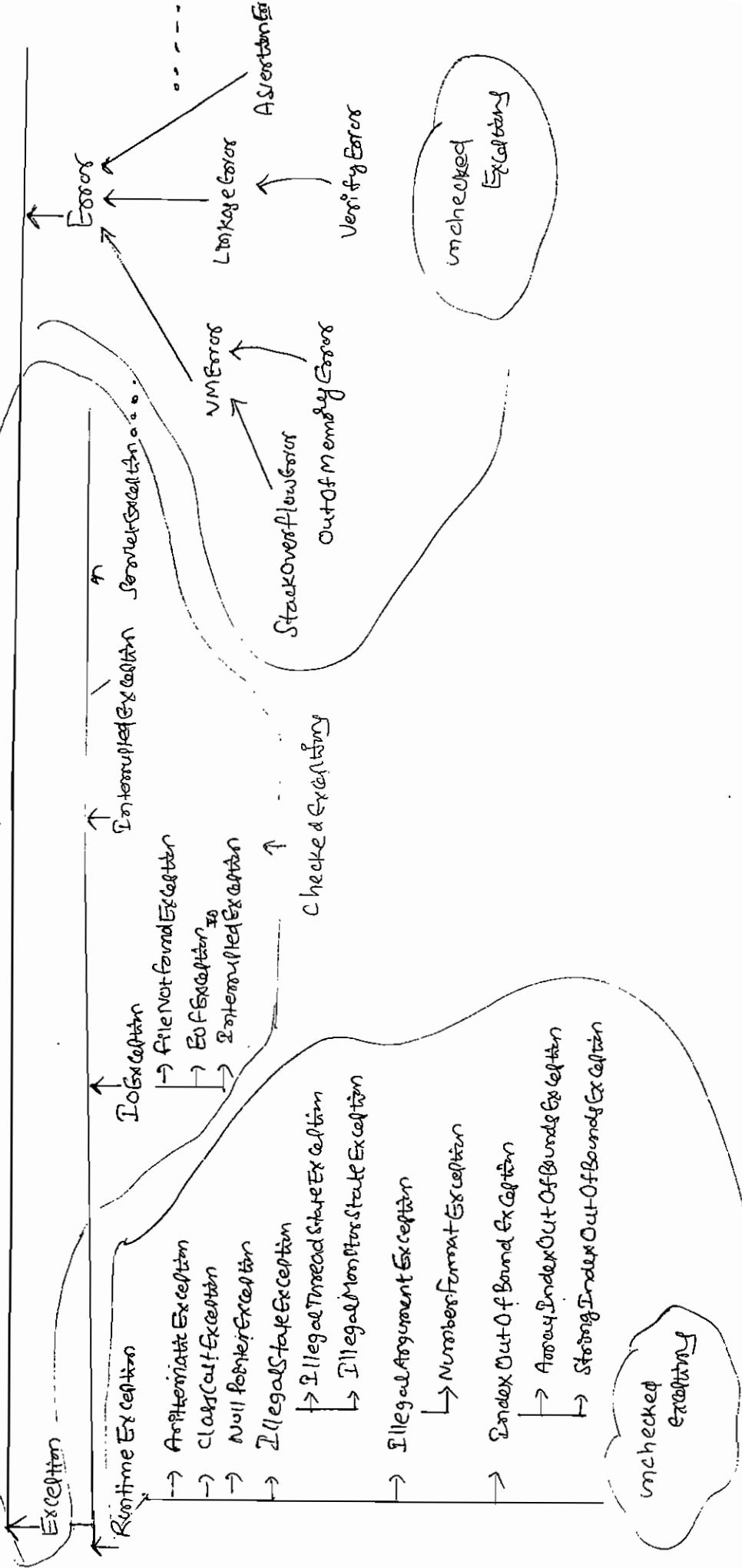
Ex: FileNotFoundException  
InterruptedException  
FileNotFoundException  
ThreadDeathException  
!!

In our program if there is any chance of raising CheckedException then compulsorily we have to handle that exception otherwise our code won't be compiled.

Eg: class Test  
{  
    public void main(String[] args)  
    {  
        Thread.sleep(5000);  
    }  
}

CSE: unreported exception "java.lang.InterruptedIOException"  
must be caught or declared to be thrown.

## Throwable



## Unchecked Exception

- (2) The Exceptions which are not checked by the compiler whether programmer is handling or not are called unchecked exceptions.

Eg: ArithmeticException

NullPointerException

BombBlastException

⋮

Class Test

↓  
Pc v main(String[] args)

{ SOP(10/0);

}

(Code compiled)

→ will print 10/0  
(exception)

- RuntimeException and its child classes, Error and its child classes are unchecked all the remaining are checked exceptions.

- Whether the exception is checked or unchecked it will be raised always at runtime and there is no chance of raising any exception at compiletime.  
(compiletime exceptions are only checked not raised)

## Fully Checked vs Partially checked Exceptions

### Fully checked

A checked exception is said to be fully checked exception iff (if and only if) all its child classes are checked.

Eg: IOException

InterruptedException

### Partially checked exceptions

- A checked exception is said to be partially checked exception iff some of its child classes are unchecked

Eg: Exception

Throwable

- The only partially checked exception in entire exception hierarchy are Exception and Throwable.

- Q) describe the nature of following exceptions

1) IOException → fully checked

2) RuntimeException → unchecked

3) InterruptedException → fully checked

4) Error → unchecked

5) Throwable → partially checked

6) ArithmeticException → unchecked

7) NullPointerException → unchecked

8) Exception → partially checked

9) FileNotFoundException → fully checked.

## 5) Customized exception handling by using try-catch

We can handle exceptions by using try-catch syntax.

```
try
{
    // Risky code
}
catch (X e)
{
    // Exception handling code
}
```

X - can be any exception

- With in the try block we have to place risky code and corresponding exception handling code should be placed in the catch block.
- The statements which might throw exceptions is called risky code

Ex 1: class Test {  
 public static void main(String[] args)  
{  
 System.out.println("stmt1");  
 System.out.println("stmt2");  
 System.out.println("stmt3");  
 }  
}

RG: Abnormal execution + 1 by zero  
Abnormal termination

Ex 2: class Test {  
 public static void main(String[] args)  
{  
 try  
 {  
 System.out.println("stmt1");  
 System.out.println("stmt2");  
 System.out.println("stmt3");  
 }  
 catch (ArithmaticException e)  
 {  
 System.out.println("10/2");  
 }  
 }  
}

g/p

stmt1  
5

Normal termination

(not recommended)

↓ even though we handle the exception it will terminate if no exception is raised

class Test {  
 public static void main(String[] args)  
{  
 System.out.println("stmt1");  
 try  
 {  
 System.out.println("10/0");  
 }  
 catch (ArithmaticException e)  
 {  
 System.out.println("10/2");  
 System.out.println("stmt3");  
 }  
 }  
}

g/p:  
stmt1  
5  
stmt3

Normal termination  
(recommended)

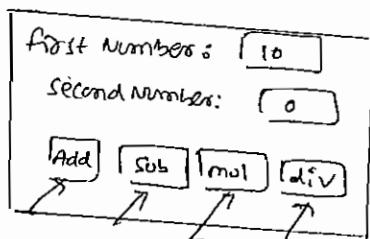
↓ This approach is recommended

This approach is recommended  
for unexpected errors  
as 10/2 is not already written in  
catch

With in the try block if anywhere an exception raised then rest of the try block won't be executed even though we handle that exception hence with in the try block we have to take only risky code but not normal code.

- The length of the try block should be as less as possible.
- Strictly speaking we should not use try-catch syntax for unchecked exceptions because for unchecked exceptions alternative path of execution won't be there.
- try-catch syntax is required only for checked exceptions because usually for checked exceptions alternative path of execution will be available.
- We can avoid unchecked exceptions (not every exception) by using conditional checkings.

Eg



To avoid AE

```

int nv = 0;
int dv = 0;
if (dv != 0)
{
    int result = nv/dv;
    // set the result into result field.
}
else
    =

```

nv -> numerator value  
dv -> denominator value

To avoid NullPointer Exception

```

if (com1 == null)
{
    com1 = new com();
}

```

Class Employee

```

{
    int id;
    String name;
    Employee(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
}
```

Public boolean equals(Object o) (e2)

```
{
    int id1 = this.id;
    String name1 = this.name;
    Employee e2 = (Employee)o;
    int id2 = e2.id;
    String name2 = e2.name;
    ...
}
```

```

Employee e1 = new Employee(101, "Sankant");
e2 = new Employee(102, "Somanta");
e3 = new Employee(103, "Sankant"); }

```

```

System.out.println(e1.equals(e2)); → false
System.out.println(e1.equals(e3)); → true
System.out.println(e1.equals("Nap")); →

```

```

if (o instanceof Employee)
{
    Employee e2 = (Employee)o;
}

```

ClassCastException

(15-08-2014)

## 6) Control flow with try-catch

```

try
{
    Stmt1;
    Stmt2;
}
    Stmt3;
Catch(X e)
{
    Stmt4;
}
    Stmt5;

```

Case 1:

If there is no Exception 1,2,3,\$ Normal termination

Case 2:

If an exception raised at Stmt2 and corresponding catch block matched  
1,1,4,5 Normal termination

Case 3:

If an exception raised at Stmt2 and corresponding catch block not matched  
\$ Abnormal termination

Case 4:

If an exception raised either at Stmt4 or at Stmt5 then it is always abnormal termination

→ The statement which is raising an exception is not part of the try block then it is always abnormal termination

## 7) Methods to print Exception information

Throwable class defines the following ③ methods to print Exception information to the console.

Method	Printable Format
printStackTrace()	Name of the exception : Description Stack Trace
toString()	Name of the exception : Description
getMessage()	Description

```

class Test {
    public static void main(String[] args) {
        try {
            System.out.println(10/0);
        }
    }
}

```

java.lang.ArithmaticException : 1 by zero  
at Test.main()

Catch (ArithmaticException e)

{ e.printStackTrace(); }

Sop(e);  $\Rightarrow$  Sop(e.toString());

Sop(e.getMessage());

java.lang.ArithmaticException : 1 by zero

1 by zero

- Default Exception handler internally uses `PrintStackTrace()` method to print exception information to the console.

## ⑧ try with multiple catch blocks

- In general exception to exception the way of handling will be carried. Hence per exception to exception we have to use separate, separate catch blocks instead of using single catch block.
- Try with multiple catch blocks is allowed and highly recommended to use.

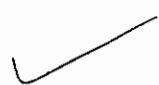
```
try
{
    // risky code
}
catch(Exception e)
{
}
not recommended
```

```
try
{
    // risky code
}
catch(SQLException e)
{
    // use mysql db instead of Oracle db
}
catch(FileNotFoundException e)
{
    // use local file to continue rest of the
    // program normally
}
```

Recommended

- ⑨ When ever we are taking try with multiple catch blocks the order of catch blocks is very important and that orders should be child to parent (top to bottom) otherwise we will get compile time error, saying (exception xxx has already been caught)

```
try
{
}
catch(ArithmeticException e)
{
}
catch(Exception e)
{
}
```



```
try
{
}
catch(Exception e)
{
}
catch(ArithmeticException e)
```



Ex: Exception `java.lang.ArithmaticException` has already been caught.

If we are writing them we will get more than one catch block for the same exception type compiletime error

```

try {
    ...
} catch (ArithmaticException e) {
}
} catch (ArithmaticException e) {
}
}

```

CSE Exception java.lang.ArithmaticException has already been caught

## Q) finally block

- It is not recommended to maintain cleanup code inside a try block because there is no guarantee for the execution of every statement within the try block.
- It is not recommended to maintain cleanup code inside catch block because if there is no exception in the try block then catch block won't be executed.
- Hence we required a special place which should be executed always irrespective of exception raised (or) not raised, handled or not handled to maintain cleanup code.

Such type of special place is nothing but finally block.

Ex:

```

try {
    open DB connection
    read data from DB
    [Close DB Connection]
} catch (X e)
{
}
}
non recommended

```

```

try {
    open DB connection
    Read data from DB
} catch (X e)
{
    [Close DB Connection]
}
not recommended

```

```

try {
    // Risky code
} catch (X e)
{
    // handling code
}
finally {
    // cleanup code
}
Recommended

```

- (a) \* The specificity of finally block is it will be executed always irrespective of exception raised or not raised, handled (or) not handled.

Ex:

```

class Test
{
    public static void main(String[] args)
    {
        try {
            System.out.println("try");
        } catch (Exception e) {
            System.out.println("catch");
        }
        finally {
            System.out.println("finally");
        }
    }
}

```

try  
finally

```

class Test
{
    public static void main(String[] args)
    {
        try {
            System.out.println("try");
        } catch (ArithmaticException e) {
            System.out.println("catch");
        }
        finally {
            System.out.println("finally");
        }
    }
}

```

AE → ArithmaticException

```

class Test
{
    public static void main(String[] args)
    {
        try {
            System.out.println("try");
        } catch (ArithmaticException e) {
            System.out.println("catch");
        }
        finally {
            System.out.println("finally");
        }
    }
}

```

NPE → null pointer exception

⑥ Even though return statement present in try and catch blocks still finally block will be executed, after that only return statement will be considered by JVM.

finally block dominates return statement

Ex:

```
Class Test
{
    public static void main(String[] args)
    {
        try {
            System.out.println("try");
            return;
        } catch (Exception e) {
            System.out.println("catch");
            return;
        } finally {
            System.out.println("finally");
            System.out.println("return");
        }
    }
}
```

C's main void main

return; and return;

return → JVM takes control

Ex 2) Class Test

```
public class Test
{
    public static void main(String[] args)
    {
        try {
            System.out.println("111");
        } catch (ArithmaticException e) {
            System.out.println("222");
        } finally {
            System.out.println("333");
        }
        String s = null;
        System.out.println(s.toString());
    }
}
```

OP → NullPointerException

⑦ Default exception handler can handle only one exception at a time and it is the most recently raised exception

Ex: Class Test {

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("111");
        public static int m()
        {
            try {
                return 777;
            } catch (Exception e) {
                return 888;
            }
        }
        finally {
            System.out.println("999");
        }
    }
}
```

OP → 999

method m() directly  
calling so write  
if static

↳ Conclusion

If return statement present in try, catch & finally blocks then finally block return statement will be considered.

18-08-2014 | Monday

- There is only one situation where finally block won't be executed and it is when ever JVM encounters System.exit(0)
- when ever JVM encounters System.exit(0) then JVM itself shutdown and there is no chance of executing any other statements including finally block

Ex:

```
class Test
{
    public static void main (String [] args)
    {
        try
        {
            System.out.println ("try");
            System.exit (0);
        }
        catch (Exception e)
        {
            System.out.println ("catch");
        }
        finally
        {
            System.out.println ("finally");
        }
    }
}
```

### System.exit(0)

The argument of exit() method represents status code.

0 → means normal termination and

non zero means abnormal termination.

→ JVM internally uses this status code to terminate the program either normally (0) or abnormally.

→ Whether it is zero or non zero we can't identify the difference and program will be terminated.

### Q) final vs finally vs finalize()

#### final

- It is a modifier applicable for classes, methods and variables.
- If a class is declared as final then we can't create child class for that class.
- If a method declared as final then overriding of that method is not possible.
- If a variable declared as final then it will become constant and we can't perform reassignment to that variable.

final class J
{
 int a = 10;
 int b = 20;
}

finalize()
{
 ...
}

finally at is a block which is always associated with try-catch and which can be used to maintain cleanup code.

- The speciality of finally block is it will be executed always irrespective of exception raised or not raised, handled or not handled.

### finalize()

It is a method present in Object class to maintain cleanup code just before destroying the object garbage collector calls finalize() method on that object to perform cleanup activity.

Once finalize() method execution completed the corresponding object will be destroyed from the memory.

Note:-

finally block is recommended to use over finalize() method because we can't expect exact behaviour of ~~finalize~~ Garbage collector, It is JVM vendor dependent (e.g.:-

### (11) Control flow in try-catch-finally

```
try
{
    Stmt1;
    Stmt2;
    Stmt3;
}
catch(X e)
{
    Stmt4;
}
finally
{
    Stmt5;
}
Stmt6;
```

Case 1: If there is no exception

Statement 1, 2, 3, 5, 6 Normal termination

Case 2: If an exception raised at Statement (2) and corresponding catch block matched

1, 4, 5, 6 normal termination

Case 3: If an exception raised at Statement (2) and corresponding catch block not matched

1, 5 abnormal termination

Case 4: If an exception is raised at statement 4 then it is always abnormal termination but before that abnormal termination finally block will be executed.

Case 5: If an exception is raised after statement 5 (or) at statement 6 then it is always abnormal termination.

Note 1:

usually exceptions will be raised in try-block but there may be a chance of raising catch and finally blocks also.

Note 2:

If any statement which is raising the exception is not part of the try block then it is always abnormal termination.

(19) Control flow on nested try-catch-finally

20-08-2014

try {

stmt1;

stmt2;

stmt3;

try

{ stmt4;

stmt5;

stmt6;

}

Catch(x e)

{ stmt7;

}

finally

{ stmt8;

}

}

Catch(x e)

{ stmt9;

}

finally

{ stmt10;

}

Case 1: if there is no exception

[1, 2, 3, 4, 5, 6, 8, 9, 11, 12, Normal termination]

Case 2:

if there is an exception raised at statement ② and corresponding catch block matched.

[1, 10, 11, 12, Normal termination]

Case 3:

if an exception raised at statement ② and corresponding catch block not matched.

[1, 11, Abnormal termination]

Case 4:

if an exception raised at statement ⑤ and inner catch block matched

[1, 2, 3, 4, 7, 8, 9, 11, 12, Normal termination]

Case 5:

if an exception raised at statement ⑤ and inner catch not matched but outer catch block matched.

[1, 2, 3, 4, 8, 10, 11, 12, Normal termination]

Case 6:

if an exception raised at statement ⑤ and both inner and outer catch blocks not matched.

[1, 2, 3, 4, 8, 11, Abnormal termination]

Case 7:

if an exception raised at statement ① and corresponding catch block matched.

[1, 2, 3, , , , , 8, 10, 11, 12, normal termination]

may be 4 may be 5 may be 6

Case 8:

if an exception raised at statement ⑦ and corresponding catch block not matched.

[1, 2, 3, , , , , 8, 11, Abnormal termination]

Case 9:

if an exception raised at statement ⑧ and corresponding catch block matched.

[1, 2, 3, , , , , , 10, 11, 12, Normal termination]

Case 10: If an exception raised at statement (9) and corresponding catch block not matched.

1, 2, 3, ., ., ., 11, Abnormal termination

Case 11B: If an exception raised at start (9) and corresponding catch block matched.

1, 2, 3, ., ., ., 8, 10, 11, 12, Normal termination

Case 12B: If an exception raised at Statement (9) and corresponding catch block not matched.

1, 2, 3, ., ., ., 8, 11, Abnormal termination

Case 13B: If an exception raised at Statement (10) then it is always abnormal termination but before that abnormal termination JVM executes finally block. i.e. Statement (11)

Case 14B:

If an exception raised at statement (11) or

statement (12) then it is always abnormal termination

(P/I a Statement which don't  
execute or not able to execute  
+)

Note:

Once control entered into try block there is no chance of coming out without executing corresponding finally block. but  
but if control not entered into try block it won't execute corresponding finally block.

(13) Various possible combinations of try-catch-finally

try { } y catch(x e) { } }	try { } y catch(x e) { } } c.e x Exception x has already been caught	try { } y catch(x e) { } } finally { } y 	try { } y catch(x e) { } } y catch(y e) { } y x 
try { } y } c.e try without catch or finally	catch(x e) { } } c.e catch without try	finally { } } catch finally with our try	

```

try
{
    finally
}

```

```

try
{
    catch(X e)
}
finally
{
}
finally {
}
c.e: finally
without toy

```

```

try
{
    finally
}
catch(X e)
{
}
c.e: catch without
toy

```

```

try
{
    finally
}
try
{
    catch(X e)
}

```

```

try
{
    sop("Hello");
    catch(X e)
}
c.e: toy without
catch or finally
c.e: catch with out toy

```

```

try
{
    catch(X e)
}
sop("Hi");
finally {
}
c.e: finally without
toy

```

```

try
{
    catch(X e)
}
finally
sop("Hello")

```

```

try
sop("Hello");
catch(X e)
}
finally
c.e: X

```

```

try
{
    catch(X e)
    sop("catch");
    finally
}

```

```

try
{
    catch(X e)
    sop("catch");
    finally
}

```

```

try
{
    catch(X e)
}
finally
sop("finally");

```

```

try
{
    try
    {
        catch(X e)
    }
    catch(X e)
}
finally

```

```

try
{
    catch(X e)
}
try
{
    catch(X e)
}
finally

```

```

try
{
    catch(X e)
}
finally
try
{
    catch(X e)
}
finally

```

in above u do  
e.g. if i = 0 gets i = 1

## Conclusion

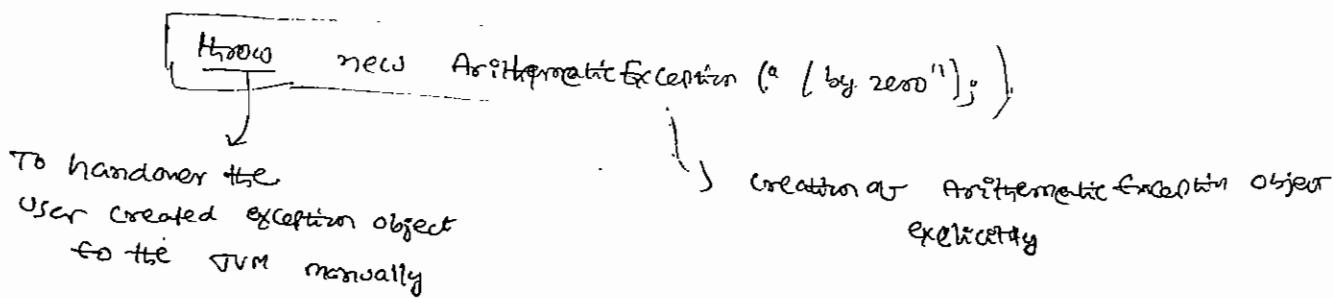
- ① The order of try-catch & finally is very important and that order must be try, catch followed by finally only.
- ② In between try and catch, catch and finally other statements are not allowed.
- ③ For try-catch-finally { } curly braces are mandatory.
- ④ Only try, only catch, only finally is not allowed.
- ⑤ try must be followed by either catch or finally.
- ⑥ We can take try-catch-finally inside try, catch and finally blocks.

(14)

## Throw keyword

Some times we can create our own exception objects

(User defined exception objects) and we can handover these exception objects to the JVM manually for this purpose we have to use throw keyword.



→ The result of following two programs is exactly same

```
Class Test
{
    public static void main(String[] args)
    {
        } for (1/0);
    }
}
```

Res: ArithmeticException: 1/ by zero  
at Test.main()

Here main() method is responsible to create ArithmeticException object and handovers to the JVM automatically

```
Class Test
{
    public static void main(String[] args)
    {
        throw new ArithmeticException("1/ by zero");
    }
}
```

Res: ArithmeticException: 1/ by zero  
at Test.main()

Here programmer is responsible to create ArithmeticException Object explicitly and handovers to the JVM manually by using throw keyword

→ If we can use throw keyword to handover our own created exception objects to the JVM manually.

Q1-08-14)

Case 1: throw e;

If e refers null then we will get NullPointerException

```
class Test
```

```
{ static ArithmeticException e = new ArithmeticException ("1 by zero"); }
```

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

```
{     throw e;
```

```
}
```

```
R.E: ArithmeticException : 1 by zero
```

```
at Test.main()
```

here class not there  
it becomes a bug because  
so far static variable can not  
accessed from other class  
in main class

```
class Test
```

```
{ static ArithmeticException e;
```

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

```
{     throw e;
```

```
}
```

```
R.E: NullPointerException
```

Case 2:

After using throw keyword we are not allowed to use any other statement otherwise we will get compilation error saying unreachable statement.

Ex:

```
class Test
```

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

```
    { System.out.println ("10/0"); }
```

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

```
}
```

```
R.E: ArithmeticException: 1 by zero  
at Test.main()
```

here AE is unreachable expression so compiler  
not able to see this line in compilation

```
class Test
```

```
{
```

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

```
{
```

```
    throw new ArithmeticException  
        ("1 by zero");
```

```
}
```

```
{
```

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

```
}
```

C.E: unreachable statement

Case 3 we can use throw keyword only for throwable types by mistake  
If we are using for non throwable type then we will get compiletime error saying incompatible types.

Ex:

```
class Test
{
    public static void main(String[] args)
    {
        throw new ArithmeticException("1 by zero");
    }
}
```

R.E: ArithmeticException : 1 by zero  
at Test.main()

class Test

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

```
{     throw new Test();
}
```

C:\ Incompatible types

found: Test

required: java.lang.Throwable

Class Test extends RuntimeException

```
{ public static void main(String[] args)
{
    throw new Test();
}

R.E: Exception in thread "main" at
Test.main()
```

class Test extends Throwable

```
{ public static void main()

```

```
{     throw new Test();
}
}

R.E: Exception in thread "main" at
Test.main()
```

Throwable

Exception

RuntimeException

Test

Case 4:

If we use throw keyword for checked exception then we will get compilation error saying unreported exception XX; must be caught (or) declared to be thrown

Ex: class Test

```
{ public static void main(String[] args)
{
    throw new Exception();
}
```

Get UnreportedException java.lang.Exception  
must be caught or declared to be thrown

class Test

```
{ public static void main()

```

```
{     throw new Error();
}
}

R.E: Exception in thread "main"
java.lang.Error
```

at Test.main()

Note we can use throw keyword for both checked and unchecked exceptions

Java APPException extends Throwable → Checked Exception

```

A.java
class A
{
    public void m() throws APPException
    {
        try
        {
            // risky code
        }
        catch (Exception e)
        {
            e.printStackTrace();
            throw new APPException();
        }
    }
}

```

```

Test.java
class Test
{
    public void main(String[] args)
    {
        A a = new A();
        a.m();
    }
    catch (APPException e)
    {
        e.printStackTrace();
    }
}

```

## (15) Throws keyword

In our program if there is any chance of raising checked exception then compulsorily we have to handle that exception otherwise we will get compiletime error saying unreported exception XXX, must be caught or declared to be thrown.

Ex

```

class Test
{
    public void main(String[] args)
    {
        Thread.sleep(5000);
    }
}

```

C:E unreported exception java.lang.Interruptedexception must be caught or declared to be thrown

We can resolve this compiletime error in two ways

① By using try-catch

```

class Test
{
    public void main(String[] args)
    {
        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
        }
    }
}

```

code compiles fine

## ② By using throws keyword

We can use throws keyword to delegate the responsibility of Exception handling to the caller (caller can be a method or JVM) then caller is responsible to handle that exception.

```
class Test
{
    public void main(String[] args) throws InterruptedException
    {
        Thread.sleep(5000);
    }
}
```

Code compiles fine

- \* Handling the exception is possible only by using try-catch and by using throws we are not handling any exception, we are just delegating the responsibility of Exception handling to the caller.
- \* Usage of throws is just to convince the compiler and it won't avoid abnormal termination of the program.
- \* Usage of throws is required only for checked exceptions and if we are using for unchecked exceptions then there is no use.

Cave 1:- / we can use throws keyword only for methods and constructors but not for classes.

Ex:-

class Test	class Test	class Test throws exception
{	{	{
public void main() throws exception	Test() throws exception	X
}	}	
3	3	Y
✓	✓	

Cave 2:-

we can use throws keyword only for throwable types otherwise we will get compilation error saying incompatible types.

class Test	class Test	class Test extends Exception
{	{	{
public void main() throws exception	public void main() throws Exception	public void main() throws Test
}	3	3
✓	X	✓

K.E. Incompatible types  
found: Test  
required: Throwable

Case 3: coith in the try block if there is no chance of raising fully checked exception then we can't write catch block for that exception otherwise we will get compiletime error saying Exception XXX is never thrown in body of Corresponding try statement.

→ This rule is not applicable for partially checked and unchecked exceptions.

Class Test

```
{ public static void main(String[] args)
{
    try {
        System.out.println("Hello");
    } catch (Exception e) {
        e.printStackTrace();
    }
}}
```

O/P: Hello

Class Test

```
{ public static void main(String[] args)
{
    try {
        System.out.println("Hello");
    } catch (IOException e) {
    }
}}
```

C.E.: ~~Exception IOException~~

is never thrown in  
body of corresponding  
try statement

Class Test

```
{ public static void main()
{
    try {
        System.out.println("Hello");
    } catch (Error e) {
    }
}}
```

O/P: Hello

Class Test

```
{ public static void main(String[] args)
{
    try {
        System.out.println("Hello");
    } catch (ArithmaticException e) {
    }
}}
```

O/P: Hello

Class Test

```
{ public static void main(String[] args)
{
    try {
        System.out.println("Hello");
    } catch (InterruptedException e) {
    }
}}
```

X

C.E.:

exception InterruptedException is  
never thrown in body or corresponding  
try statement

case 4 class Test

```

    {
        public void main(String[] args)
        {
            throw new Exception();
        }
    }

```

```

class Test
{
    public void main(String[] args) throws Exception
    {
        throw new Exception();
    }
}

```

C.E: Unreported exception.  
Exception; must be caught or declared to be thrown

```

class Test
{
    public void main(String[] args)
    {
        throw new Error();
    }
}

```

R.E: exception in thread "main" java.lang.Error  
at Test.main()

Note 5 we can use throw, throws keywords and catch block only for the throwable types otherwise we will get compiletime error saying incompatible types.

Ex: Class Test

```

class Test
{
    public void main(String[] args)
    {
        try
        {
            // ...
        } catch(Test e)
        {
            // ...
        }
    }
}

```

C.E: Incompatible types

found: Test  
required: Throwable

```

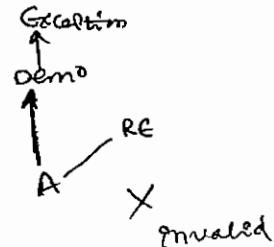
Runtimeexception
class Test extends Throwable
{
    public void m()
    {
        try
        {
            // ...
        } catch(Test e)
        {
            // ...
        }
    }
}

```

Note 6 we can create our own exceptions as either checked or unchecked and there is no chance of creating user defined exception ~~base~~ as partially checked.

because multiple inheritance is not supported by

Java through classes



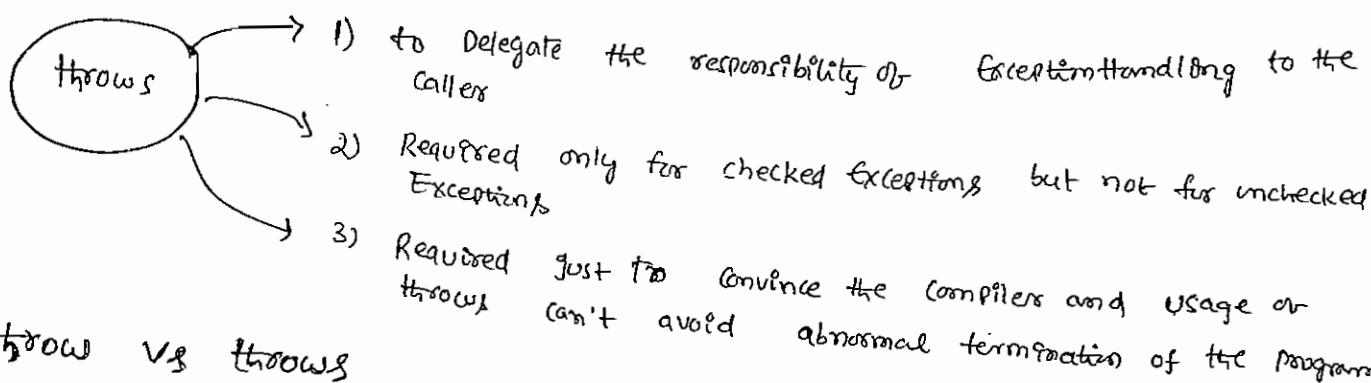
```

class Test extends Exception
{
    public void main(String[] args)
    {
        try
        {
            catch(Test e)
            {
                X;
            }
        }
    }
}

```

C-E exception test is never thrown  
in body of corresponding try statement

↑ In the above example Test class is extending Exception class so that it will become checked exception and there is no chance of creating a child class for Test which is unchecked hence Test will become fully checked exception i.e. the reason why we are getting compiletime error for the above example



### throw vs throws

- We can use throw keyword to handover user created exception object to the JVM manually
- We can use throws keyword to delegate the responsibility of exception handling to the caller.

### (17) Exception Handling keywords summary

- ① try : To maintain risky code
- ② catch : To maintain exception handling code
- ③ finally : To maintain cleanup code
- ④ throw : To handover our own created exception to the JVM manually
- ⑤ throws : To delegate the responsibility of exception handling to the caller

## (17) Various possible compiletime errors in Exception Handling

- ① Undeclared exception xxx ; must be caught or declared to be thrown
- ② exception xxx has already been caught
- ③ try without catch or finally
- ④ catch without try
- ⑤ finally without try
- ⑥ incompatible types
  - found: xxx
  - required: Throwable
- ⑦ unreachable Statement
- ⑧ exception xxx is never thrown in body of corresponding try

## (18) Customized Exceptions (User defined Exceptions)

25-08-14

Some times we can create our own exceptions, such type of exceptions are called customized exceptions.

Ex:

TooOldException

TooYoungException

SleepingException

If we want to create our own exceptions as unchecked then our exception class is extended from RuntimeException.

If we want to create our exceptions as checked exceptions (fully checked) then our exception should be extended from either Exception or Throwable. We can't create our own exceptions as partially checked exceptions.

Ex:

```
class TooYoungException extends RuntimeException
```

```
{  
    TooYoungException (String s)  
}
```

↳ Main class → .java file

```
y  
class TooOldException extends RuntimeException
```

```
{  
    TooOldException (String s)  
}
```

↳ Super(s);

y  
y

```

class Test
{
    public static void main (String [] args)
    {
        int age = Integer.parseInt (args [0]);
        if (age <= 18)
            {
                throw new TooYoungException ("your age is already crossed  

                    marriage age..... no chance of getting marriage");
            }
        else if (age > 60)
            {
                throw new TooOldException ("wait for some more time...  

                    You will get best match soon.");
            }
        else
            {
                System.out ("you will get match details through email---!");
            }
    }
}

```

→

```

class Test
{
    public static void main (String [] args) throws InterruptedException
    {
        foo();
    }

    public static void foo() throws InterruptedException
    {
        bar();
    }

    public static void bar() throws InterruptedException
    {
        Thread.sleep (5000);
    }
}

```

exception raised at a method  
at least we must handle it  
in caller?

In the above example If we remove ~~at least one~~ throws keyword then code won't compile.

### Exception Propagation

If a method is raising any exception then that method is responsible to handle it.

If that method is not handling then that exception will be propagated to the caller then caller is responsible to handle it this is called Exception Propagation

## Class Test

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            foo();
        }
        catch (ArithmaticException e)
        {
            System.out.println("catch");
        }
        public static void foo()
        {
            System.out.println("foo");
        }
    }
}

```

Output  $\Rightarrow$  Catch

unchecked exception not required to use throws

## Exception Wrapping

- Sometimes after handling the exception we can wrap that exception information into another exception (most of the cases application specific exception) and we will throw that exception  
 $\rightarrow$  This is called exception wrapping.

Ex8

```

class AppException extends RuntimeException
{
    AppException(RuntimeException e)
    {
        super(e);
    }
}

class Test {
    public static void main(String[] args)
    {
        try
        {
            System.out.println("foo");
        }
        catch (ArithmaticException e)
        {
            throw new AppException(e);
        }
    }
}

```

## class Test

```

public static void main(String[] args)
{
    try
    {
        foo();
    }
    catch (InterruptedException e)
    {
        System.out.println("catch");
    }
    public static void foo() throws InterruptedException
    {
        Thread.sleep(5000);
    }
}

```

Output  $\Rightarrow$  no output

just spec waiting time.

## Cause

Exception in thread "main".

AppException: java.lang.ArithmaticException:  
 $/$  by zero at Test.main()

Caused by: E.T. ArithmaticException:  
 $/$  by zero  
at Test.main()

Wrapping



## (19) Top-10 Exceptions

Based on the person who is raising the exception all the exceptions are categorized into two types

- ① JVM Exceptions
- ② Programmatic Exceptions

### ① JVM Exceptions

The exceptions which are raised automatically by JVM whenever an event occurs are called JVM Exceptions

Ex: ArithmeticException  
NullPointerException ...

### ② Programmatic Exceptions

The exceptions which are raised explicitly either by the programmer or by API developer are called programmatic exceptions

Ex: IllegalArgumentException  
TooYoungException  
;

## Top-10

### ① ArrayIndexOutOfBoundsException :

- ① It is the child of RuntimeException and hence it is unchecked.
- ② It will be raised automatically by JVM whenever we are trying to access an array element with ~~out of~~ index range value.

Ex:  
int[] a = {10, 20, 30};

Sop(a[1]); → 20

Sop(a[10]); → R.E ⇒ ArrayIndexOutOfBoundsException

### ② NullPointerException

It is the child of RuntimeException and hence it is unchecked. It will be raised automatically by JVM whenever we are trying to access some method call on null reference.

Ex:  
String s = null;  
Sop(s.tostring()); → R.E ⇒ NullPointerException

### ③ ClassCastException

- It is the child of RuntimeException and hence it is unchecked.
- It will be raised automatically by JVM whenever we are unable to perform typecasting from one type to another type.

Ex:

Object obj = new String ("Srikanth");

StringBuffer sb = (StringBuffer) obj;

Result ClassCastException

String can not be cast to  
StringBuffer

valid cast

Ex 2:

Object obj = loger(o);

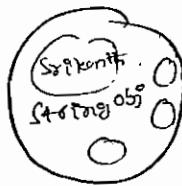
obj.substring(3); X

If (obj instanceof String)

d String s = (String) obj; ✓

Srikanth

String object



Last object ↴

String subname = s.substring(3);

for (i); → // Kanth

↳

26-08-2014

### ④ NoClassDefFoundError

- It is the child class of Error and hence it is unchecked.
- It will be raised automatically by JVM whenever we are trying to run a class and that class .class file is not available.

Ex:

java Test ↴

If Test.class file is not available then we will get

NoClassDefFoundError (until 1.6 m'y)

Ex:

class Test

{ p s v main (String[] args)

d Demo d = new Demo();

≡

) }

at the time of running Test class if Demo.class file is not available then we will get NoClassDefFoundError

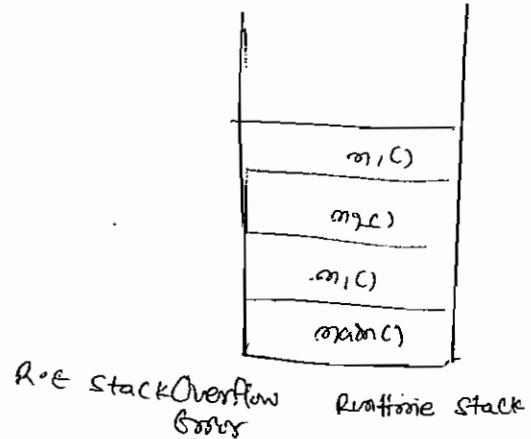
## 5) StackOverflowError

- It is the child class of Error and hence it is unchecked.
- It will be raised automatically by JVM whenever we are trying to perform recursive method invocation.

Ex: class Test

```

    {
        public void main(String[] args)
        {
            m1();
        }
        public static void m1()
        {
            m2();
        }
        public static void m2()
        {
            m1();
        }
    }
  
```



## 6) ExceptionInInitializerError

- It is the child class of Error and hence it is unchecked.
- It will be raised automatically by JVM whenever an exception raise while executing static variable assignments and static blocks.

Ex:

```

    class Test
    {
        static int x=10/0;
    }
  
```

R.E: ExceptionInInitializerError  
Caused by: ArithmeticException / by zero

{ /> static { } , main();  
  |    | ;  
  |    | static { } ;  
  |    | ;  
  |    | Searching wise definable. Lexically twice  
  |    | no definition. }

class Test

```

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

R.E: ExceptionInInitializerError  
Caused by: NullPointerException

## 7) IllegalArgumentExeption

It is the child class of RuntimeException and hence it is unchecked.  
It will be raised explicitly either by the programmer or by API developer to indicate that a method has been invoked with illegal argument.

Ex: The allowed values range for a thread priority is 1 - 10 &  
by mistake if we are trying to set any other priority then we will get  
IllegalArgumentExeption

Thread t = new Thread();

```

    t.setPriority(10); ✓
    t.setPriority(100); } } R.E
    t.setPriority(0); } } IllegalArgumentExeption
  
```

Dateformat df = DateFormat.getDateTimeInstance(0, 0);  
Dateformat df = DateFormat.getDateTimeInstance(4, 4); → R-E  
because for Date and Time the only allowed  
values range is 0 to 3

6

## NumberFormatException

- ↳ It is the child class of **IllegalArgument Exception**, which is child of **RuntimeException** and hence it is unchecked.
  - ↳ It will be raised explicitly either by the programmer or by API developers to indicate that we are trying to convert String to primitive and that string is not representing number.

Στα

```
int x = Integer.parseInt("10");
int y = Integer.parseInt("ten");
```

## RG: Numbers from Ex 3.2.1

If it is developed by positive action as well as by interpretation

6

## Illegal State Exception

It is the child class of RuntimeException and hence it is unchecked. It will be raised explicitly either by the programmer (or) by API developer to indicate that a method has been invoked at runtime.

१५

Once we started a thread we are not allowed to restart the same thread again otherwise we will get `RuntimeException` saying `IllegalThreadStateException`

eg: Thread t = new Thread();  
t.start();  
;;;  
t.start(); R.E Blrg

t.start(); R.E IllegalThreadStateException

Ex2: once a session got expired (invalidated) we can't call any method on that session object otherwise we will get Runtime exception saying IllegalStateException

### ⑩ AssertionError

- It is the child class of Error and hence it is unchecked.
- It will be raised explicitly either by the programmer or by API developer to indicate that assert statement fails.

Ex: assert (b);

↓  
boolean type

If assert statement fails then we will get AssertionException

Eg: Class Test

↓ P r o c e s s m a i n ( s t r i n g [ ] a r g s )

↓ int x=10;

    " " "

assert (x>10);

    ";";;

s o l ( x );

JAVA EXCEPTION

java -ea Test

Re: AssertionException

## Summary

Exception or Error	Raised By
① ArrayIndexOutOfBoundsException ② NullPointerException ③ ClassCastException ④ NoSuchElementException ⑤ StackOverflowError ⑥ ExceptionInInitializerError	Raised automatically by JVM and hence these are JVM Exceptions
⑦ IllegalArgumentException ⑧ NumberFormatException ⑨ IllegalStateException ⑩ AssertionError	Raised explicitly either by the programmer or by API developer and hence these are programmer exceptions

## 1.7 Version Enhancements

In 1.7 version as part of Exception-handling the following concepts introduced.

- (1) try with resources
- (2) multi-catch block

### (1) try with resources

In traditional try-catch-finally all the resources should be declared outside the try block and whatever the resources we are opening as part of try block all those resources should be closed explicitly by the programmer inside finally block before closing the resources even we required to perform null checking also to avoid NullPointerException.

```
import java.util.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Connection con=null;  
        Statement st=null;  
        ;;;;  
        try  
        {  
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
            System.out.println("Driver Loaded");  
            con=DriverManager.getConnection("jdbc:odbc:oradbc",  
                "System", "durga");  
            System.out.println("Connection established");  
            st=con.createStatement();  
        }  
        ;;;;  
        catch(Exception e)  
        {  
            e.printStackTrace();  
        }  
        finally  
        {  
            try  
            {  
                if (con!=null)  
                    con.close();  
                if (st!=null)  
                    st.close();  
            }  
            ;;;  
        }  
    }  
}
```

Catch(~~Exception~~ e)

```
{ e.printStackTrace();
}
```

)  
The problems in the above approach are

- (1) finally block is mandatory to close the resources whatever we opened as part or try block. for every resource null checking is mandatory before closing it.
- (2) with this length of the code will be increased and hence readability and performance will be reduced.
- (3) programmer should perform all these activities explicitly so that complexity of the programming will be increased.  
→ to overcome the above problems sun people introduced try with resources concept in 1.7 version.

(27-07-2014)

According to this concept we can declare a try block with the required resources

Syntax:

```
try (Resource)
{
    // Risky code
    ...
    catch(X e)
    {
        // Handling code
    }
}
```

→ we can declare multiple Resources also but these resources should be separated with Semicolon (;

Syntax:

```
try (R1; R2; R3)
{
    ...
    catch(X e)
    {
        ...
    }
}
```

```

import java.sql.*;
class Test {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection(url, user, pwd)) {
            Statement st = con.createStatement();
            for (String established);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

The main advantages of try with resources are

- ① Whatever the resources we opened as part of the try block will be closed automatically once control reaches end of the try block either normally or abnormally.
- ② We are not required to write finally block and hence length of the code and complexity of the programming will be reduced.

### Conclusion

- ① In try with resources every resource should be AutoCloseable resource  
A resource is said to be AutoCloseable iff the corresponding class implements AutoCloseable interface  
• AutoCloseable Interface present in java.lang package, introduced in 1.7 version and defines only one method close() method.  
• Public Interface AutoCloseable  
    Positive abstract void close() throws j.l.Exception
- ② Every resource reference variable is by default final hence we can't perform reassignment to that reference variable.

Ex:

```

try (BufferedReader br = new BufferedReader(new FileReader(
        "C:\\Windows\\System32\\cmd.exe"))) {
    // use br to read data from the file
    // and it will be closed automatically once
    // control reaches end of the try block either
    // normally or ...
}

```

`br=new BufferedReader(new FileReader("idig.txt"));` → c.e. 185  
 auto-closable resource  
~~br~~ may not be assigned

```

  }
  {
  }
  }
```

(3) If we are applying try with resources for non auto-closable resources then we will get compilation error

Ex:

```

try (Test t=new Test())
{
}
}
```

c.e.s  
 try-with-resources not applicable to  
 variable type (~~t=new Test()~~)  
 Required: AutoCloseable  
 Found: Test

Note: (1)

Until 1.6v finally block is mandatory to close the resources but from 1.7 version onwards to close the resources if we are using

finally block is not required  
try with resources

Note: (2)

until 1.6v try must be followed by either catch or finally but from 1.7v onwards only try-with-resources allowed without catch block also.

```

try(R)
{
}
}
```

✓ from 1.7v onwards

javac -source 1.6 Test.java

(2) Multi-catch block

until 1.6v even though multiple exception types are having same handling code we required to write a separate catch block

Ex:

```

try
{
}
}

try (ArithException e)
{
  e.printStackTrace();
}

try (NullPointerException e)
{
  e.printStackTrace();
}
```

Same code

e.printStackTrace();

Catch (NullPointerException e)
 {
 e.printStackTrace();
 }
 }

e.printStackTrace();

```

    Some Code
    {
        Catch (InterruptedException e)
        {
            System.out.println(e.getMessage());
        }
        Catch (FileNotFoundException e)
        {
            System.out.println(e.getMessage());
        }
    }
}

```

- The problem with this approach is length of the code will be increased and hence readability & performance will be reduced.
- To overcome the problem Java people introduced multi-catch block in 1.7 version
- According to TIPS concept we can declare a single catch block with multiple exception types

Ex:

```

try
{
    ...
}
Catch (ArithmaticException | NullPointerException e)
{
    e.printStackTrace();
}
Catch (InterruptedException | FileNotFoundException e)
{
    System.out.println(e.getMessage());
}
}

```

In multi catch block exception types should not be related with any relationship (either child to parent or parent to child) otherwise we will get compile time error

Ex:

```

try
{
    ...
}
Catch (Exception | ArithmaticException e)
{
    ...
}

```

```

try
{
    ...
}
Catch (ArithmaticException | Exception e)
{
    ...
}

```

☞ Alternatives in multi-catch statement can not be related by subclassing

## Assertions (1.4v)

- ① Introduction
- ② assert as a keyword & Identifier
- ③ Types of assert statements
- ④ Various Possible Runtime flags
- ⑤ Appropriate & Inappropriate use of assertions
- ⑥ Assertion Errors

### ① Introduction

- Very common way of debugging is usage of `sop` statements but the problem with `sop`'s is after fixing the bug compulsorily we have to delete `sop` statements. otherwise these `sop`'s will be executed at runtime sever logs.
- To overcome this problem sun people introduced Assertions concept in 1.4 version
- The main advantage of Assertions when compared with `sop`'s is after fixing the bug we are not required to remove `Assert` statements because they won't be executed by default at runtime based on our requirement we can enable & disable assertions and by default assertions are disabled.
- Hence the main objective of assertions is to perform debugging usually we can perform debugging in Development (dev) and Test environments but not in production environment hence Assertions concept applicable for development and test environments but not for production.

### (2) assert as keyword & Identifier

`assert` keyword introduced in 1.4 version hence from 1.4 version onwards we can't use `assert` as identifier otherwise we will get compiletime error

Ex: class Test

```

d
    s u main (String[] args)
    {
        int assert = 10;
        sop (assert);
    }
y
  
```

javac Test.java ↴ C:\C

as in delete 1.4, 'assert' is a keyword and may not be used as an identifier (use -source 1.3 or lower to use assert as identifier?)

2) `javac -source 1.3 Test.java`

→ Code compiles fine but with warning

3) `java Test`

QP: 10

✓ `javac -source 1.2 Test.java`

✓ `javac -source 1.3 Test.java`

✗ `javac -source 1.4 Test.java`

✗ `javac -source 1.5 Test.java`

Note:

- (1) If we are using assert as identifier and if we are trying to compile according to old version (1.3 or lower) then the code compiles fine but with warning.
- (2) We can compile a java program according to a particular version by using `-source` option

(3)

### Types of assert statements

There are

(2) types of assert statements

(1) simple version

(2) augmented version

(1) Simple version

Syntax

`assert(b);`

'b' should be boolean type

→ If b is true then our assumption satisfied & hence rest of the program will be executed normally.

→ If b is false then our assumption fails i.e. somewhere something goes wrong and hence the program will be terminated abnormally by raising Assertion Error.

Once we got Assertion Error we will analyze the code & we can fix the problem.

Ex 6

Class Test

{

    public void main(String[] args)

{

        int x=10;

}

        assert(x>10);

    }

`javac Test.java`

`java Test  
QP: 10  
java -ea Test`

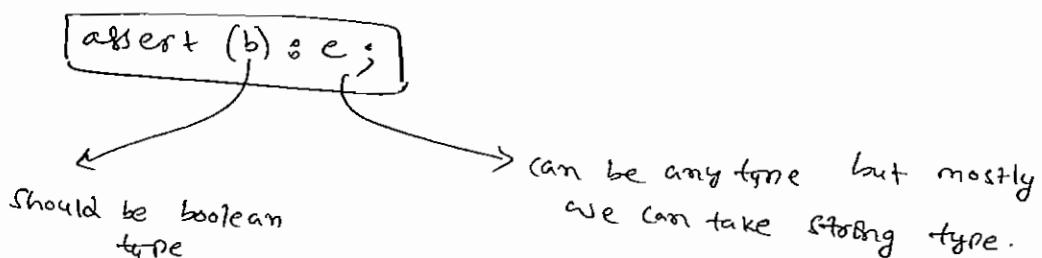
Result: Assertion error

Note By default assert statements won't be executed because assertions are disabled by default but we can enable assertions by using -ea option

## (2) Augmented Version

We can argument some description with AssertionError by using augmented version

Syntax



Ex

```
class Test
{
    public static void main(String[] args)
    {
        int x=10;
        !!!
        assert (x>10); "Here x value should be >10 but it is not";
        !!!
        System.out.println(x);
    }
}
```

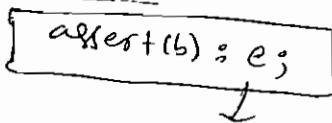
javac Test.java ✓

java Test ✓  
O/P: 10

java -ea Test

Re: AssertionError: Here x value should be  
> 10 but it is not

Conclusion (1)



e will be executed iff (if and only if) first argument is false i.e if the first argument is true then second argument won't be evaluated.

```
class Test
{
    public static void main(String[] args)
    {
        int x=10;
        !!!
        assert (x==10) : ++x; // x=11
        !!!
        System.out.println(x);
    }
}
```

javac Test.java

java Test ←  
O/P: 10

java -ea Test

O/P: [10]

## Conclusion 2

`assert(b) : e;`

for the second argument we can take method call but void return type method call is not allowed. otherwise we will get compiletime error.

Ex: class Test

```

    {
        public static void main(String[] args)
        {
            int x=10;
            assert(x>10) : m();
            System.out.println(x);
            public static int m()
            {
                return 777;
            }
        }
    }
  
```

javac Test.java

java Test

Output: 10

java -ea Test

Result: Assertion Error: 777

if m() method return type is void then we will get compiletime error saying 'void' type not allowed here.

Note 8 Among all versions of asserting it is recommended to use augmented version because it provides more information for debugging.

04/09/14

## ④ Various possible Runtime flags

① -ea | -enableassertions:

② -da | -disableassertions:  
To enable assertions for every non system class (our own classes)

③ -esa | -enableSystemAssertions:  
To disable assertions in every non system class

④ -dsa | -disableSystemAssertions:  
To enable assertions in every system class ( predefined classes)

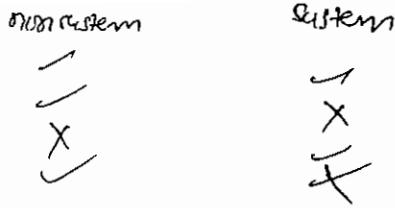
To disable assertions in every system class.

Note:

we can use above flags simultaneously then JVM will considered these flags from left to right.

Ex:

java -ea -esa -ea -dse -da -esa -ea -dse Test



at the end Assertions will be enabled in every non-system class and disable in every system class.

### Cafe Study

1) To enable assertions only in B class

`java -ea:Pack1.B`

Pack1

— A.class  
— B.class

Pack2

— C.class  
— D.class

2) To enable assertions in both B and D classes

`java -ea:Pack1.B -ea:Pack1.Pack2.D`

3) To enable assertions for every class of pack1

`java -ea:Pack1...`

4) To enable assertions in every class of pack1 except B class

`java -ea:Pack1... -da:Pack1.B`

5) To enable assertions in every class of Pack1 except Pack2 classes

`java -ea:Pack1... -da:Pack1.Pack2...`

Note: we can enable and disable assertions either class wise or package wise.

### Appropriate and Inappropriate use of assertions

① It is always inappropriate to mix programming logic with assert statement, because there is no guarantee for the execution of assert statement always at runtime.

Ex:

```
public void withdraw(double amount)
{
    if (amount < 100)
        throw new IllegalRequestException();
    else
        process request;
```

A appropriate way to use

```
public void withdraw(double amount)
{
    assert (amount >= 100);
    process request;
```

In an appropriate way

② while performing debugging in our program if there is any place where the control is not allowed to reach i.e. the best place to use assertions.

≡

Switch(x) → should be a valid month number

case 0 : `for ("JAN");  
break;`

case 1 : `for ("FEB");  
break;  
|  
|`

case 12 : `for ("DEC");  
break;  
default:`

}

assert (false); → Rer AssertionError

③ It is always inappropriate for validating public method arguments by using assertions because outside person doesn't aware whether assertions are enabled or disabled in our system.

④ It is always appropriate for validating private method arguments by using assertions because local person can aware whether assertions are enabled or disabled in our system.

⑤ It is always good appropriate for validating command line arguments by using assertions because these are arguments to main() method, which is public.

public static void main(String args){  
if (args.length > 1)  
assert (args[0].equals("durga"));

## C assert

1. Don't  $z=59$

public void m1(int x)

2. assert ( $x > 10$ ); → <sup>In appropriate</sup> ①

switch (x)

3. case 10:

System.out.println("10");  
break;

case 11: System.out.println("11");  
break;

4. } default: assert (false); → <sup>Irresponsible</sup> ②

private void m2 (Don't x)

5. assert ( $x < 10$ ); → <sup>Inappropriate</sup> ③

private void m3()

6. assert ( $m > 0$ ); → <sup>Encapsulation</sup> ④

private boolean m4()

7. {  $z = 6;$   
return true; }

05-09-14

## SCJP

① Class one

1. public class One

2. {

3. public void main (String[] args)

4. {

5. int assert = 10;  
6. System.out.println(assert);

7. }

8. }

9. class Two

10. {

11. public void main (String[] args)

12. {

13. int x = 10;  
14. assert (x > 0);

15. }

① javac --source 1.3 one.java

(Compiles fine but with warning)

② javac --source 1.4 one.java → GE

③ javac --source 1.3 Two.java  
↳ GE

④ javac --source 1.4 Two.java

(Compiles fine without warning)

## ② class Test

```

d
public static void main (String [] args)
{
    boolean assertion = false;
    assert (assertion) : assertion = true;
    if (assertion)
        {
            System.out.println ("assertion");
        }
}

```

If assertions are not enabled ?

No o/p

If assertions are enabled,

RE: AssertionError : true

## ③ class Test

```

d
public static void main (String [] args)
{
    boolean assertion = true;
    assert (assertion) & assertion = false;
    if (assertion)
        {
            System.out.println ("assertion");
        }
}

```

If assertions are not enabled:

"If : assertion"

If assertions are enabled

o/p : assertion

## ⑥ AssertionErrors

→ It is the child class of Error and hence it is unchecked. If assert statement fails (i.e. argument is false) then we will get Assertion Error.

→ Even though it is legal to catch Assertion error but it is not a good programming practice.

Ex:

```

class Test
{
    public static void main (String [] args)
    {
        int x = 10;
        ...
        ...
        try
        {
            assert (x > 0);
        }
        catch (AssertionError e)
        {
            System.out.println ("I am stupid b'z i am
                               catching AssertionErrors.");
        }
    }
}

```

← (bcz debugging concept is spoiled here)

### Note

In the case of web application if we run java program in debug mode automatically assert statements will be executed

(25-08-14)

## Garbage Collection

- (1) Introduction
- (2) The ways to make an object eligible for GC
- (3) The methods for requesting JVM to run GC
- (4) Finalization

### (1) Introduction

→ In old languages like C++ programmer is responsible to create new object and to destroy useless objects.  
→ Usually programmers taking very much care while creating objects and neglecting destruction of useless objects. Because of his negligence at certain point for creation of new object sufficient memory may not be available (because total memory filled with useless objects only) and total application will be down with memory problems. Hence OutOfMemoryError is very common problem in old languages like C++.  
→ But as Java programmer is responsible only for creation of objects and programmer is not responsible to destroy useless objects.  
Even people provided one assistant to destroy useless objects this assistant is always running in the background (demon thread) and destroy useless objects just because of this assistant the chance of failing java program with memory problems is very very low, this assistance is nothing but Garbage Collector.

Hence the main objective of Garbage Collector is to destroy useless objects

### (2) The ways to make an object eligible for GC

Even though programmer is not responsible to destroy useless objects it is highly recommended to make an object eligible for GC if it is no longer required.

An object is said to be eligible for GC if and only if it doesn't contain any reference variable.

The following are various ways to make an object eligible for GC

#### (1) Nullifying the reference variable

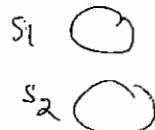
If an object no longer required then assign null to all its reference variables then that object automatically eligible for Garbage Collection this approach is nothing but nullifying the reference variable.

Nullifying the reference variable

Student s<sub>1</sub> = new Student();

Student s<sub>2</sub> = new Student();

no object eligible for GC



one object eligible for GC

s<sub>1</sub> = null;

two objects eligible for GC

s<sub>2</sub> = null;

### (ii) Reassigning the reference variable

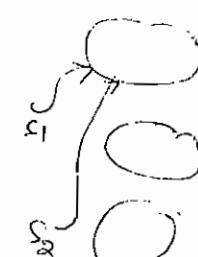
If an object no longer required then reassign its reference variable to some other object then old object by default eligible for Garbage Collection.

Reassigning the reference variable

Student s<sub>1</sub> = new Student();

Student s<sub>2</sub> = new Student();

No object eligible for GC



one object eligible for GC

(s<sub>1</sub> = new Student());

two objects eligible for GC

~~s<sub>2</sub> = new Student()~~

(s<sub>2</sub> = s<sub>1</sub>);

### (iii) Objects created inside a method

The objects which are created inside a method are by default eligible for GC once method completed

Class Test

Two objects eligible for GC

P = & main(String[] args)

d → m1();

P = & m1()

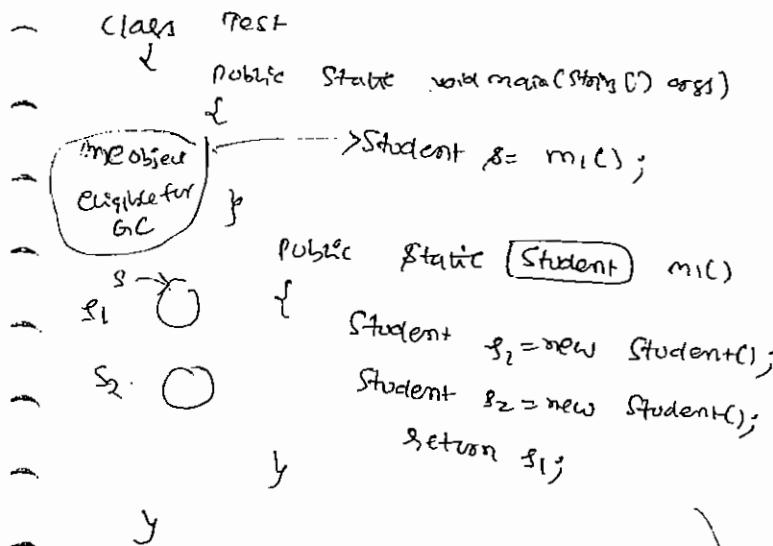
Student s<sub>1</sub> = new Student();

Student s<sub>2</sub> = new Student();

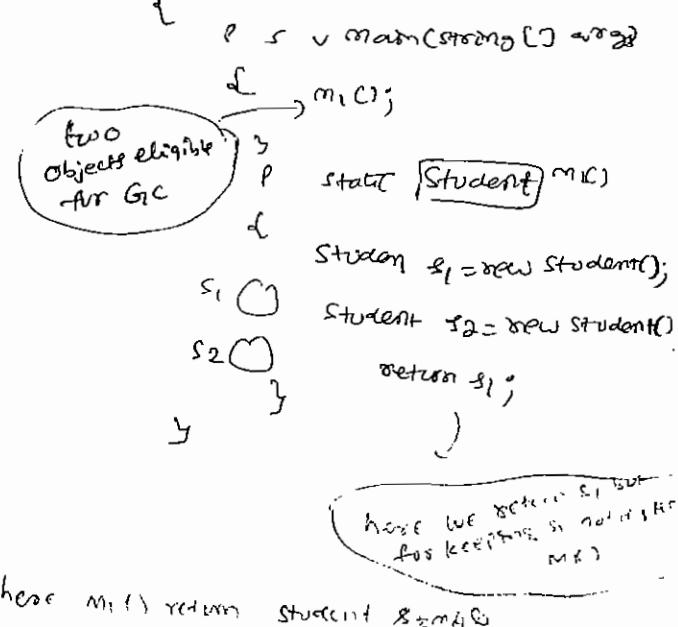


here s<sub>1</sub>, s<sub>2</sub> are still available as long as after completion of method

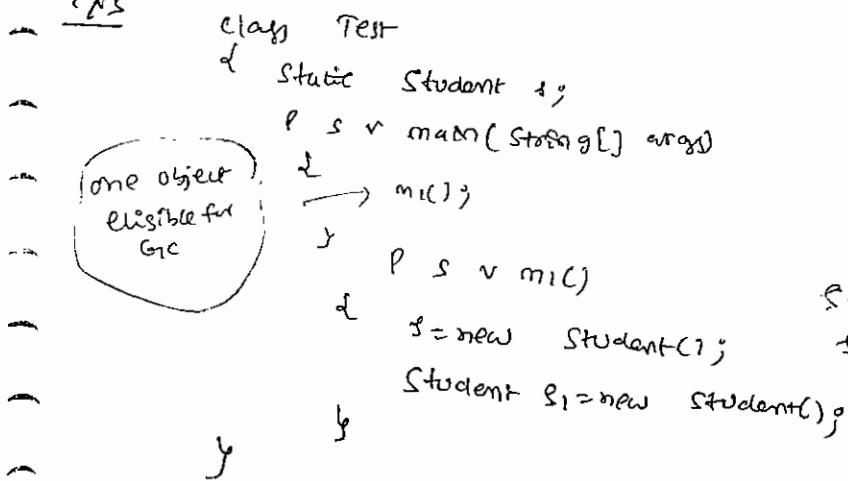
Ex2)



class Test



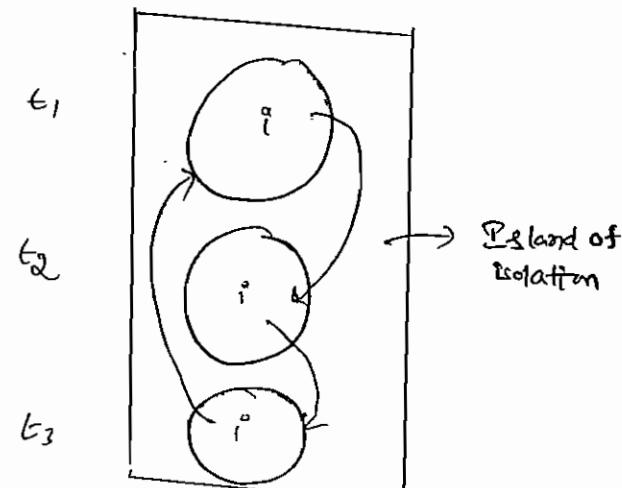
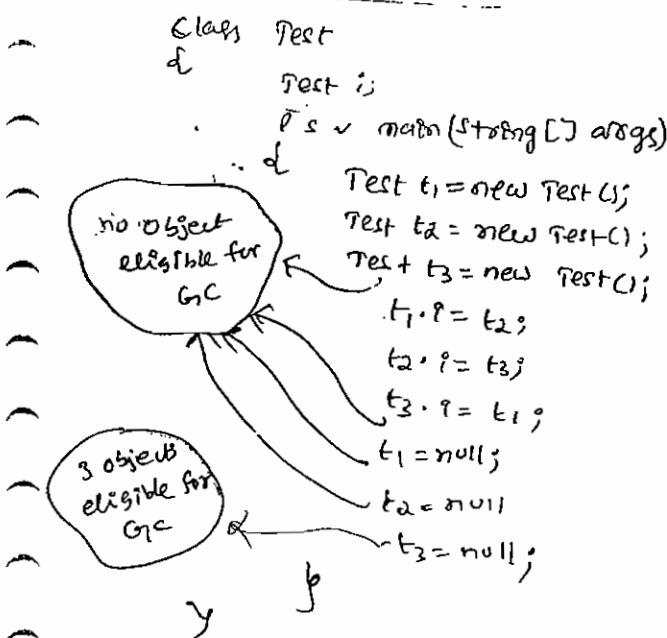
Ex3



here  $s_1$  object forced to  
method to check condition  
the method to be eligible  
for GC

(26-08-14)

### Island of Isolation



### Note

- ① If an object doesn't contain any reference variable then it is eligible for Garbage Collection always.
- ② Even though object having references sometimes it is eligible for Garbage Collection. (If all references are internal references)

Ex: Island of Isolation

③

### The ways for requesting JVM to Run Garbage Collector

- Once we made an object eligible for GC it may not be destroyed immediately by Garbage Collector. whenever JVM runs GC then only the objects will be destroyed but when exactly JVM runs Garbage collector we can't expect which it is varied from JVM to JVM.
- Instead of waiting until JVM runs Garbage Collector we can request our request or not here is no guarantee but most of the times JVM accept our request.

The following are 2 ways for requesting JVM to run Garbage Collector

① By using System class:

System class contains a static method gc(); for this purpose

System.gc();

② By using Runtime class:

- Java application can communicate with JVM by using Runtime object
- Runtime class present in `java.lang` package
- we can create Runtime object by using Runtime.getRuntime() method

Runtime r = Runtime.getRuntime();

- once we got Runtime object we can call the following methods on it

① totalMemory()

It returns number of bytes of total memory present in the Heap (i.e. Heap size)

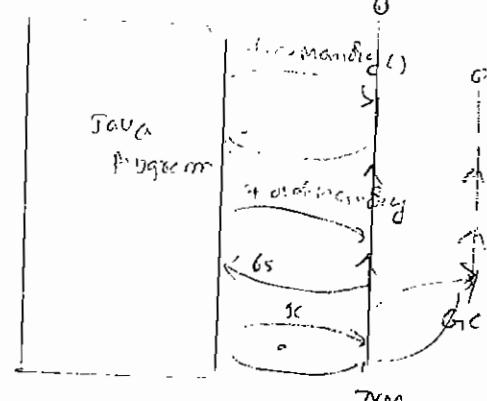
## (2) freeMemory()

It returns number of bytes of freeMemory() present in the Heap

## (3) gc() for requesting JVM to run garbage collector

Ex:

```
import java.util.Date;  
class RuntimeDemo  
{  
    public static void main(String[] args)  
    {  
        Runtime r = Runtime.getRuntime();  
        System.out.println("TotalMemory()"); 5177344  
        System.out.println("freeMemory()"); 4945200  
        for (int i=0; i<10000; i++)  
        {  
            Date d = new Date();  
            d=null;  
        }  
        System.out.println("freeMemory()"); 4714464  
        r.gc();  
        System.out.println("freeMemory()"); 5059352  
    }  
}
```



→ note:

gc() method present in System class is a static method whereas gc() method present in Runtime class is instance method.  
(1) which of the following is valid way for requesting JVM to run garbage collector.

X (1) Runtime.gc(); // instance method not occur with class name

Y (2) (new Runtime()).gc(); // for singleton class we don't create object

(3) Runtime.getRuntime().gc();

Note:

(1) It is convenient to use System class gc() method when compared with Runtime class gc() method

(2) With respect to performance it is highly recommended to use Runtime class gc() method when compared with System class gc() method because System class gc() method internally calls Runtime class gc() method.

class System

{  
 public static void gc()  
}

y [Runtime.getRuntime().gc();]

y

27-08-14

## Finalization

Just before destroying an object garbage collector calls finalize() method to perform cleanup activities.

Once finalize() completes automatically garbage collector destroys that object.

finalize() method present in Object class with the following declaration:

Protected void finalize() throws Throwable

We can override finalize() method in our class to define our own cleanup activities.

### Example

Just before destroying an object Garbage collector calls finalize() method on the object which is eligible for GC then the corresponding class finalize() method will be executed.

For example: If strong object eligible for GC then String class finalize() method will be executed but not Test class finalize() method.

Ex:

```
Class Test
{
    public void main(String[] args)
    {
        String s=new String("durga");
        s=null;
        System.gc();
        System.out.println("End of main");
    }
}
```

```
public void finalize()
```

```
{ System.out.println("finalize method called"); }
```

In the above example String object eligible for GC and hence String class finalize() method got executed which has empty implementation and hence the output is

O/P: End of main

If we replace String object with Test object then Test class finalize() method will be executed. In this case the output is

End of main  
finalize method called

(or)

finalize method called  
End of main

Case 2 Based on our requirement we can call finalize() method explicitly then it will be executed just like a normal method call and object won't be destroyed.

Exe

```

Class Test
{
    public void main(String[] args)
    {
        Test t = new Test();
        t.finalize();
        t.finalize();
        t=null;
        System.gc();
    }
    public void finalize()
    {
        System.out.println("Finalize method called");
    }
}
  
```

→ In the above program finalize() method got executed 3 times in that 2 times explicitly by the programmer and one time by the Garbage Collector.

In this case output of finalize() method is:

{  
 → finalize method called  
 → finalize method called  
 → End of main  
 → finalize method called

- If we are calling finalize() method explicitly then it will be executed like a normal method call and object won't be destroyed.
- If Garbage Collector calls finalize() method then object will be destroyed.

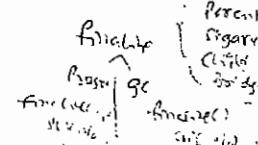
Note

init(), service(-,-) and destroy() methods are considered as lifecycle methods of servlet. Just before destroying servlet object webcontainer calls destroy() method to perform cleanup activities. But based on our requirement we can call destroy() method from init() & service() methods then destroy() method will be executed just like a normal method call and servlet object won't be destroyed.

Case 3

If programmer calls finalize() method and while executing the finalize() method if an exception raised which is uncaught then the program will be terminated abnormally by raising that exception.

If Garbage collector calls finalize() method and while executing finalize() method if an exception raised which is uncaught then JVM ignores that exception & rest of the program will be executed normally.



### Ex8 class Test

```
{ public static void main (String [] args)
{
    Test t = new Test ();
    t.finalize (); →①
    . t=null;
    System.gc ();
    System.out.println ("End of main");
}
public void finalize ()
{
    System.out.println ("finalize method called");
    System.out.println ("10%");
}
```

- ⇒ If we are not commenting line ① then programmer calls finalize() method and while executing that finalize() method ArithmeticException raised which is uncaught hence the program will be terminated abnormally by raising that exception.
- ⇒ If we are commenting line ① then GarbageCollector calls finalize() method & while executing this finalize method the an exception raised which is uncaught & hence JVM ignores that exception and rest of the program will be continued normally. In this case output is
- End of main  
finalize method call  
finalize method call  
End of main  
(or)

Which of the following is valid?

- ① JVM ignores Every exception which is raised while executing finalize() method X
- ② JVM ignores only uncaught exception which are raised while executing finalize() method /

28-06-14

Even though object eligible for GC multiple times  
but garbage collector calls finalize() method only once

### Class FinalizedDemo

```
{ static FinalizedDemo s;
{
    s = new FinalizedDemo ();
    s.main (String [] args) throws InterruptedException
    {
        FinalizedDemo f = new FinalizedDemo ();
        f.hashCode (); // 100
    }
}
```

```

f=null;
System.gc();
Thread.sleep(5000);
sop(s.hashCode()); //100
if s==null;
System.gc();
Thread.sleep(10000);
sop("End of main");
public void finalize()
{
    sop("finalize method called");
}
s=this; // gc won't destroy object
        s=null;

```

Q/P6

25724761

finalize method called

25724761

End of main method

In the above program even though object eligible for GC @ times but Garbage collector calls finalize() method only once.

Case 4:

We can't expect exact behaviour of Garbage Collector It is carried from JVM to JVM hence for the following questions we can't provide exact answers.

- ① When exactly Gar JVM runs Garbage Collector
- ② In which order Garbage Collector identifies eligible objects
- ③ ~~④~~ In which order Garbage Collector destroys eligible objects
- ④ Whether Garbage Collector destroys all eligible objects or not
- ⑤ What is algorithm followed by Garbage Collector etc....

Note:

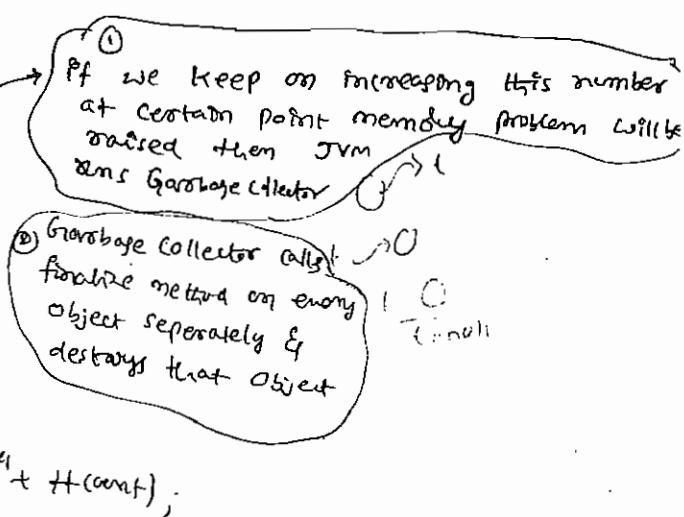
- ① Whenever program runs with low memory then JVM runs Garbage Collector but we can't expect exactly at what time.
- ② Most of the Garbage Collector follows standard algorithm Mark & Sweep algorithm. It doesn't mean every Garbage Collector follows the same algorithm.

Ex:

```

class Test
{
    static int count=0;
    public void main(String[] args)
    {
        for(int i=0; i<(10); i++)
        {
            Test t = new Test();
            t=null;
        }
        public void finalize()
        {
            sop("finalize method called: " + ++count);
        }
    }
}

```



## Case 5: Memory leaks

- The objects which are not using in our program and which are not eligible for GC such type of useless objects are called memory leak.
- In our program if memory leaks present then the program will be terminated by raising OutOfMemoryError. Hence if an object no longer required it is highly recommended to make that object eligible for GC.

The following are various third party memory management tools to identify memory leaks

- HP OVA
- HP JMeter
- JProbe
- Patrol
- IBM Tivoli etc...

# OOPS

- (1) Data hiding
- (2) Abstraction
- (3) Encapsulation
- (4) Tightly encapsulated class
- (5) IS-A Relationship } inheritance
- (6) Has-A Relationship }
- (7) method signature
- \* (8) overloading }
- \* (9) overriding }
- \* (10) static control flow
- \* (11) instance control flow
- \* (12) Constructors
- (13) Coupling
- (14) Cohesion
- (15) Type-casting

module-1

## ① Data Hiding

outside person can't access our internal data directly (or)  
our internal data should not go out directly this OOP feature is

nothing but Data hiding.

After validation or authentication outside person can access our internal data

Ex: 1) After providing proper username and password we can able to access our gmail inbox information

Ex: 2) Even though we are valid customers of the bank we can able to access our account information and we can't access others account information

By declaring Data members (~~variable~~) as private we can achieve Data hiding

```

Ex: Public class Account
    {
        private double balance;
        Public double getBalance()
        {
            / validation
            return balance;
        }
    }
  
```

The main advantage of Data hiding is security

Note: It is highly recommended to declare data members (variable) as private

(2)

## Abstraction

Hiding internal implementation and just highlight the set of services what we are offering, is the concept of Abstraction

e.g.

Through Bank ATM GUI screen Bank people are highlighting the set of services what they are offering without highlighting internal implementation.

The main advantages of Abstraction are

- ① We can achieve security because we are not highlighting our internal implementation
  - ② Without affecting outside person we can able to perform any type of changes in our internal system and hence easier enhancement will become easy
  - ③ It improves maintainability of the application
  - ④ It improves easiness to use our system
- By using Interfaces and abstract classes we can implement abstraction.

(3)

## Encapsulation

The process of binding data and corresponding methods into a single unit is nothing but encapsulation

e.g.

Class Student

{

data members

+

(behaviour) methods

}

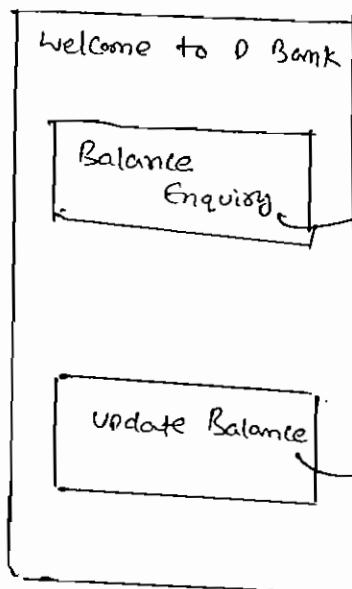


capsule.

If any component follows Data hiding and Abstraction such type of component is said to be encapsulated component.

Encapsulation = Data hiding + Abstraction

Ex:



GUI Screen

Public class Account

{ private double balance;

Public double getBalance();

// validation

return balance;

Public void setBalance(double balance);

// validation

this.balance = balance;

The main advantages of encapsulation are

- ① We can achieve security
- ② Enhancement will become easy
- ③ It improves maintainability of the application

The main advantage of encapsulation is we can achieve security but the major disadvantage of encapsulation is it increases length of the code and slows down execution.

#### (4) Tightly Encapsulated Class

A class is said to be tightly encapsulated iff (If and only if) each and every variable declared as private whether class contains corresponding getter and setter method or not and whether these methods are declared as public or not these things we are not required to check.

Ex:

Public class Account

{ private double balance;

Public double getBalance();

return balance;

a) Which of the following classes are tightly encapsulated

Ex:

✓ class A  
  { private int x=10;  
  }  
  class B extends A  
  {  
  }  
✗ class C extends A  
  {  
  private int z=30;  
  }

b) Which of the following classes are tightly encapsulated

Ex:

✗ class A  
  {  
  int x=10;  
  }  
  class B extends A  
  {  
  }  
    private int y=20;  
  }  
  class C extends B  
  {  
  }  
    private int z=30;  
  }

Note:

If the parent class is not tightly encapsulated then no child class is tightly encapsulated



## 5) IS-A Relationship

- ① It is also known as inheritance
- ② The main advantage of IS-A relationship is code reusability
- ③ By using extends keyword we can implement IS-A relationship

Ex:

```
class P
{
    public void m1()
    {
        System.out.println("Parent");
    }
}

class C extends P
{
    public void m2()
    {
        System.out.println("Child");
    }
}
```

class Test

{ P s v main(String[] args)

{ ① P p = new P();

p.m1(); ✓

p.m2(); ✗

CE: Cannot find symbol  
Symbol: method m2()  
Location: class P

{ ② C c = new C();

c.m1(); ✓

c.m2(); ✗

\* ③ P p1 = new C();

p1.m1(); ✓

p2.m2(); ✗

y P

{ ④ C c1 = new P();

↑

CE: Incompatible types  
found: P  
required: C

### Conclusions

- ① whatever methods Parent has by default available to the child and hence on the child reference we can call both Parent and child class methods
- ② whatever methods child has by default not available to the parent and hence on the parent reference we can't call child specific methods
- \* ③ Parent reference can be used to hold child object but by using that reference we can't call child specific methods but we can call the methods present in Parent class.
- ④ Parent reference can be used to hold child object but child reference can not be used to hold parent object.

### Coupling in Inheritance

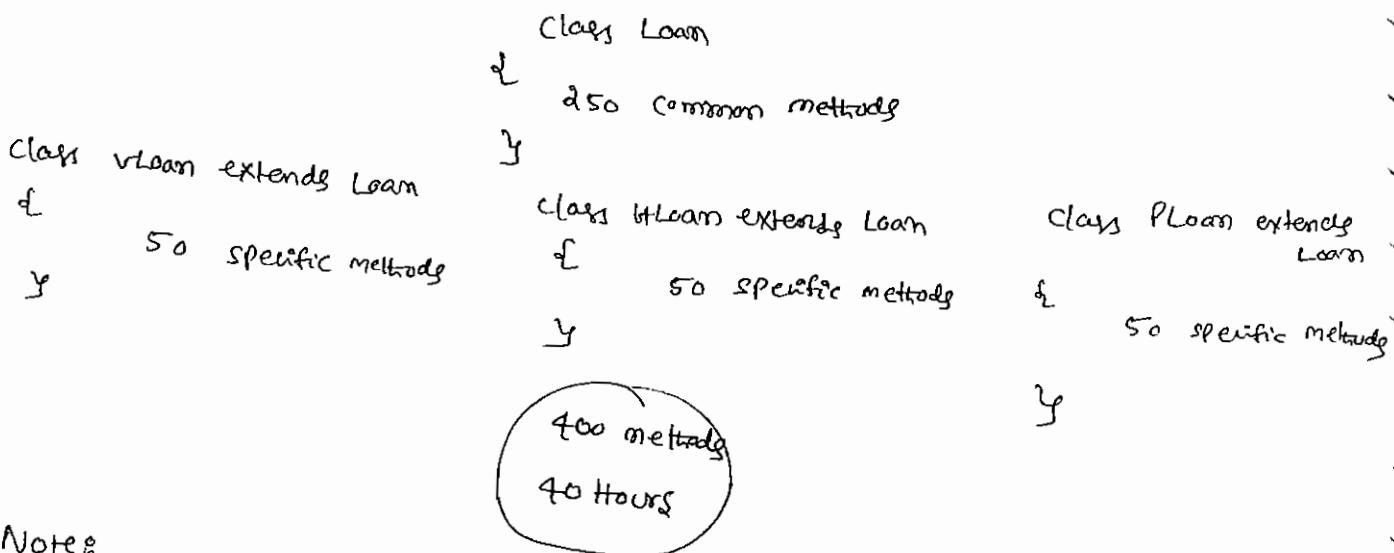
```
class VLoan
{
    300 methods
}
```

```
class HLoan
{
    300 methods
}
```

```
class PLoan
{
    300 methods
}
```

900 methods  
90.0% hours

## with inheritance

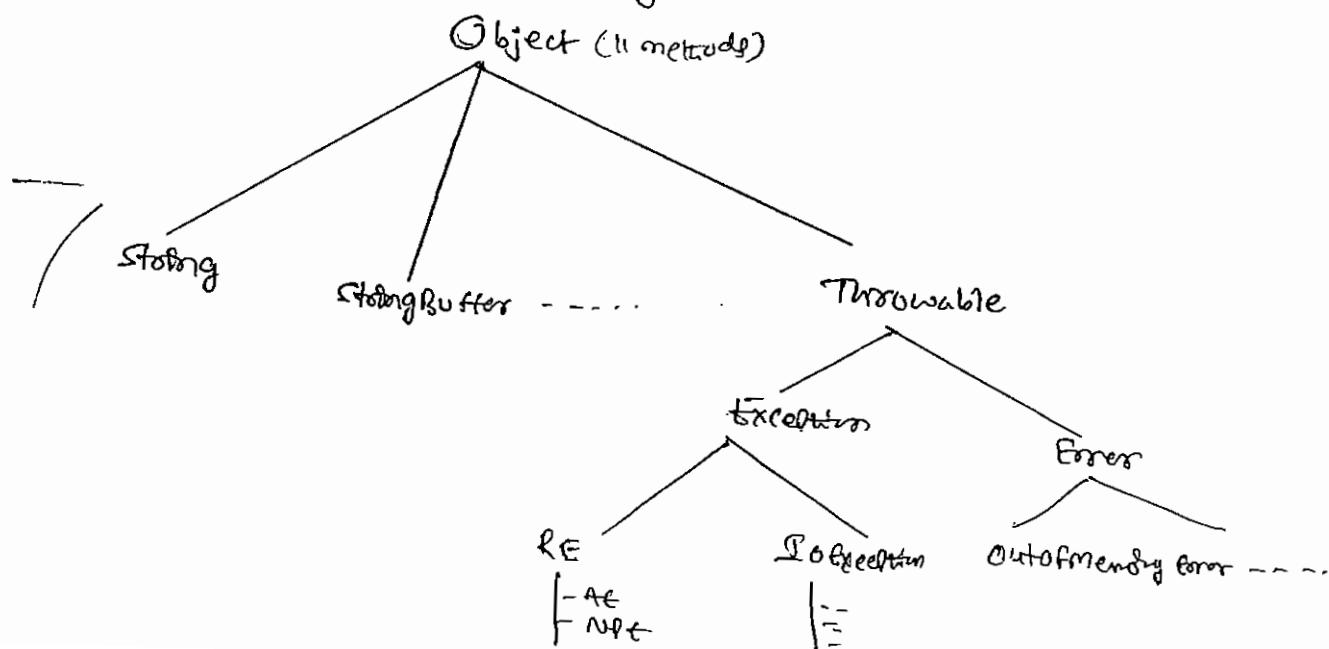


### → Notes

The most common methods which are applicable for any type of child, we have to define in Parent class

The specific methods which are applicable for a particular child we have to define in Child class.

- Total Java API is implemented based on Inheritance concept
- The most common methods which are applicable for any Java object are defined in Object class and hence every class in Java is the child class of Object either directly or indirectly so that Object class methods by default available to every Java class without re-writing due to this Object class acts as root for all Java classes.
- Throwable class defines the most common methods which are required for every Exception and Error classes hence this class acts as root for Java Exception Hierarchy



## multiple Inheritance

A Java class can't extend more than one class at a time hence Java won't provide support for multiple inheritance in classes.

Ex:

Class A extends B, C

{

} X CEE

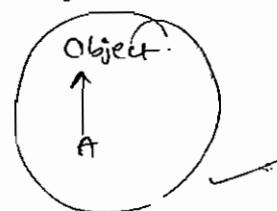
## Notes

- ① If our class doesn't extend any other class then only our class is direct child class of Object.

Ex: Class A

{

}



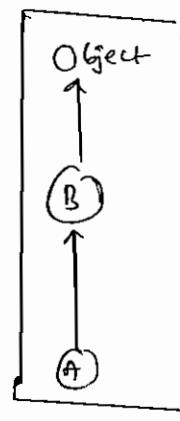
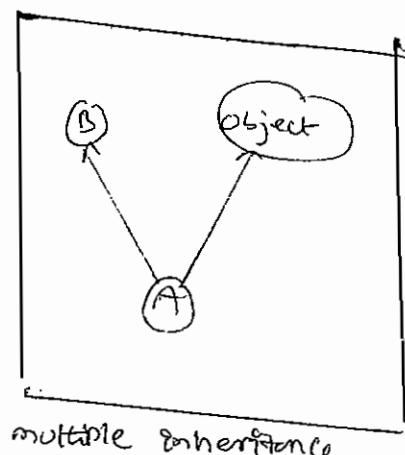
- ② If our class extends any other class then our class is indirect child class of Object.

Ex:

Class A extends B

{

}



multilevel inheritance

## Notes

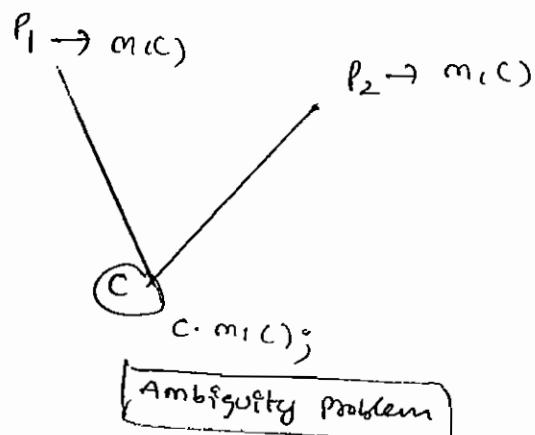
either directly or indirectly Java won't provide support for inheritance with respect to classes.

why



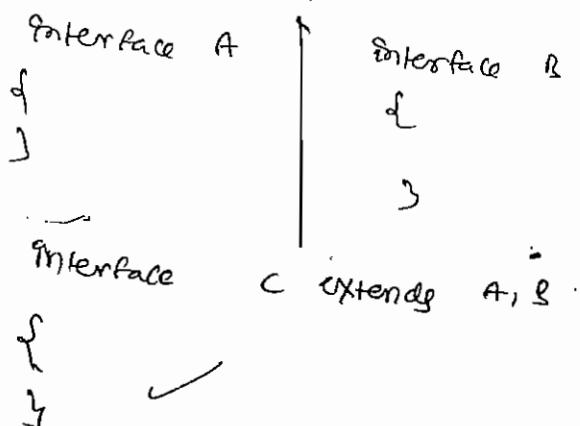
Q) why java can't provide support for multiple inheritance

There maybe a chance of ambiguity problem hence java won't provide support for multiple inheritance

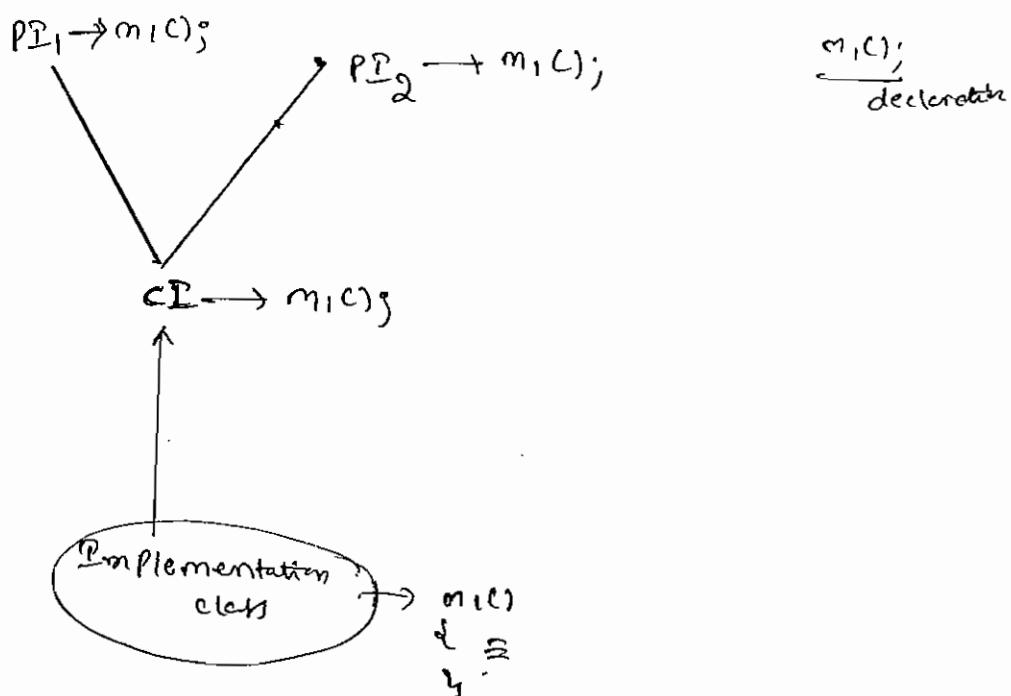


But interface hence java can extend any number of interfaces simultaneously with respect to interfaces.

Ex:



Q) why Ambiguity problem won't be there in interfaces



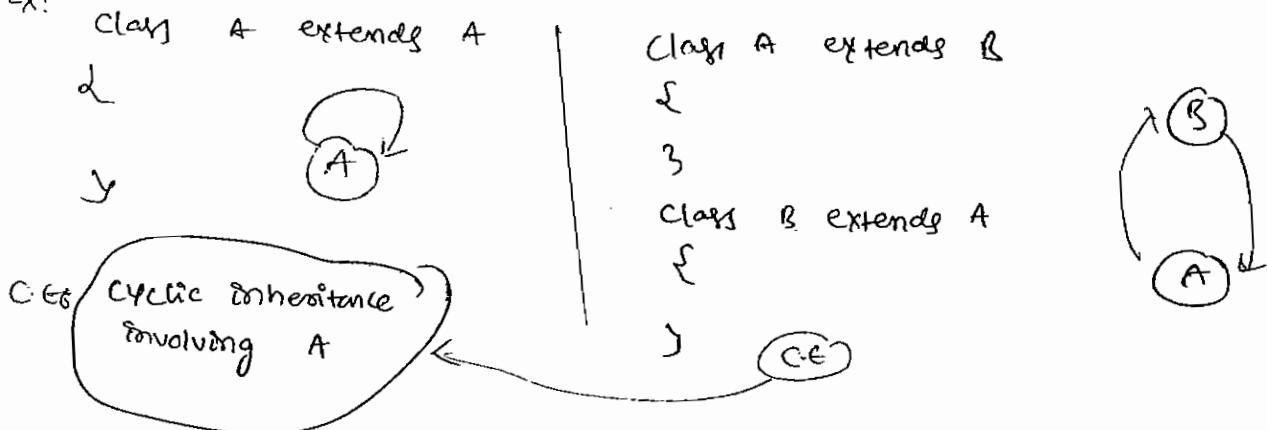
Even though multiple method declarations are available but implementation is unique and hence there is no chance of ambiguity problem in interface.

Note: Strictly speaking through interfaces we won't get any inheritance (because only declaration no implementation)

### Cyclic Inheritance

Cyclic inheritance is not allowed for java of course it is not required.

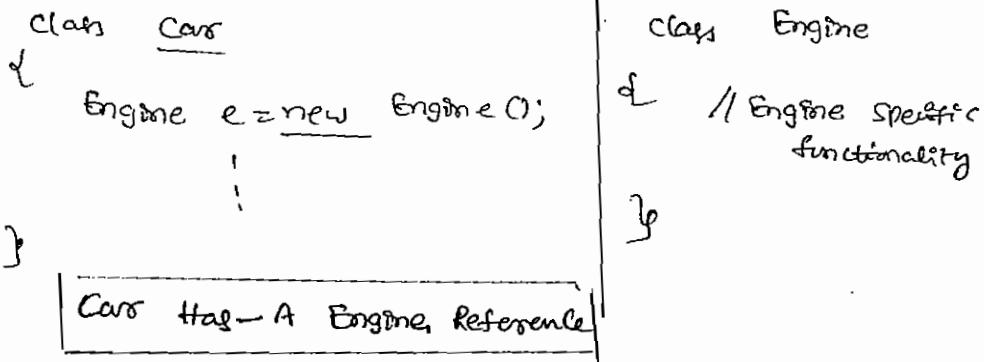
Ex:



### (6) Has-A Relationship

- (1) Has-A Relationship is also known as composition or aggregation.
- (2) There is no specific keyword to implement Has-A relation but most of the times we are depending on new keyword.
- (3) The main advantage of Has-A Relationship is reusability of the code.

Ex:



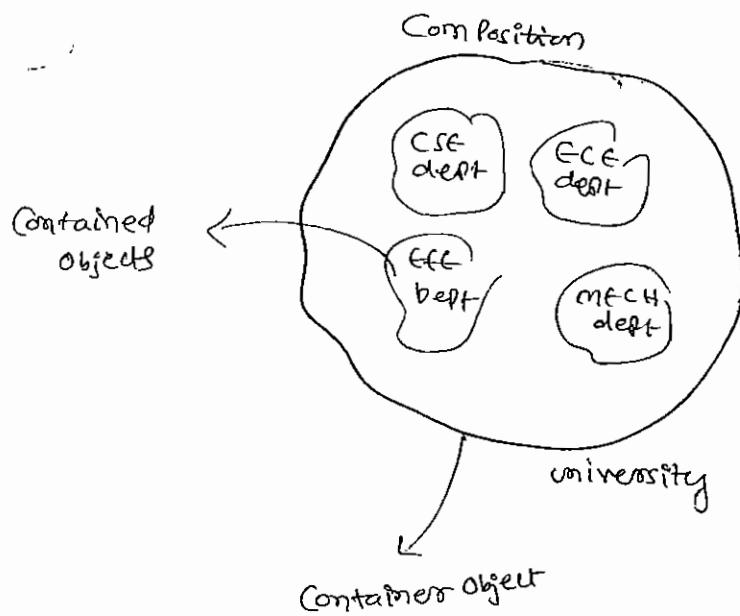
### (2) difference b/w Composition and Aggregation

#### (1) Composition

Without existing container object if there is no chance of existing contained objects then container & contained objects are strongly associated and this strong association is nothing but Composition.

Ex University consists of several departments

Without existing University there is no chance of existing department hence university and department are strongly associated and this strong association is nothing but composition.

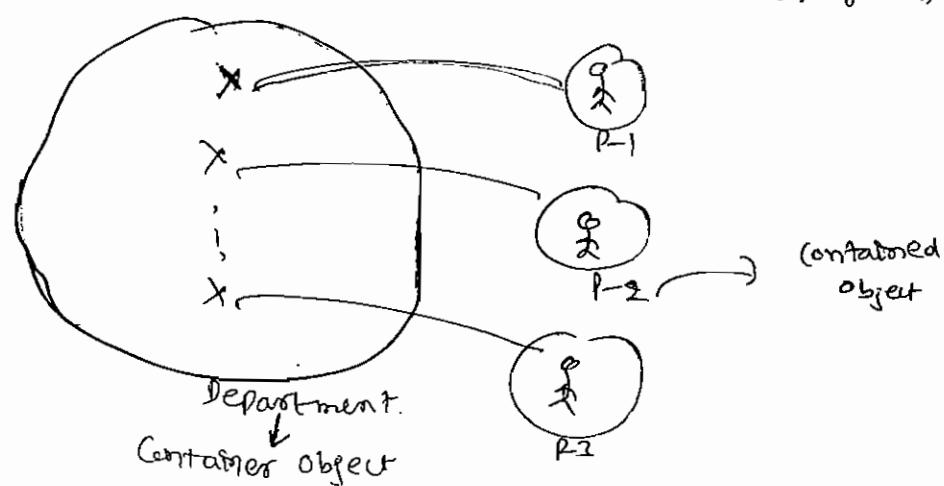


## Aggregation

Without existing Container object If there is a chance of existing contained object then container and contained objects are weakly associated and this weak association is nothing but Aggregation

Ex

Department consists of several professors. Without existing department there may be a chance of existing professor objects hence department and professor objects are weakly associated and this weak association is nothing but Aggregation



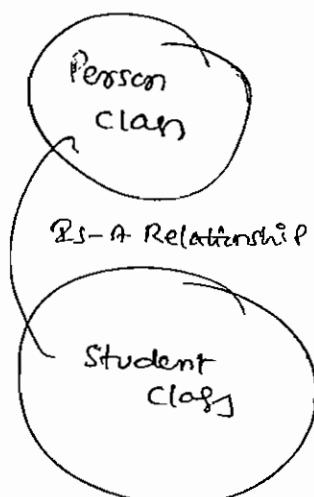
Note b ① In composition objects are strongly associated where as in aggregation objects are weakly associated

② In composition container object holds directly contained objects whereas in aggregation container object holds just references of contained objects

### Is-A vs Has-A

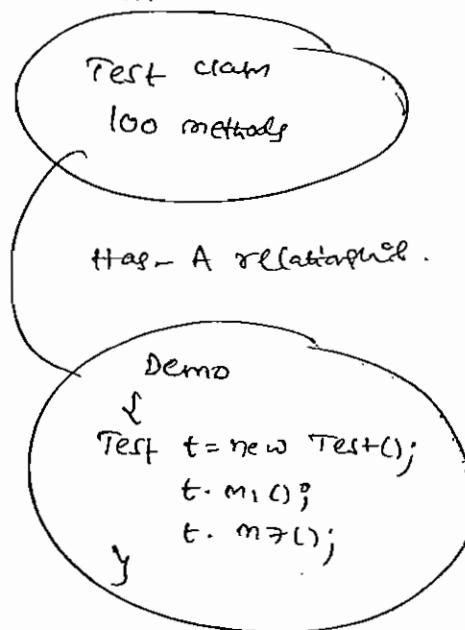
If we want total functionality of a class automatically then we should go for Is-A relationship.

Ex:



If we want part of the functionality then we should go for Has-A relationship.

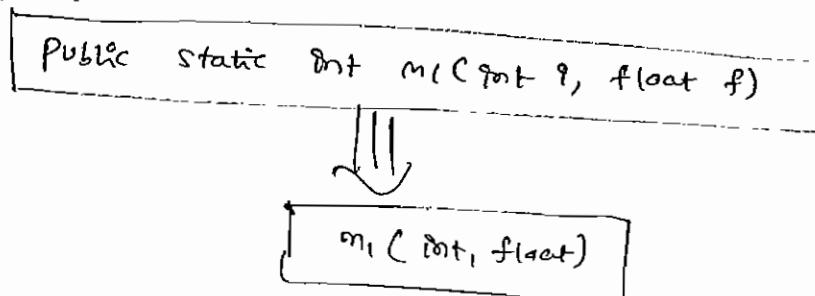
Ex:



## 7 method signature

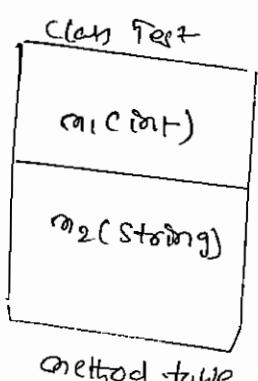
- Java method signature consists of method name followed by argument types

Ex:



- Return type is not part of method signature in Java
- Compiler will use method signature to resolve method calls

Ex 1)



```
class Test
{
    public void m1(int i)
}
public void m2(String s)
}
```

Test t = new Test();

t.m1(10);

t.m2("durga");

t.m3(10.5);

CSE

(cannot find symbol)

Symbol: method m3(double)

Location: class Test

- with in a class two methods with the same signature not allowed

Ex:

```
class Test
{
    public void m1(int i) { m1(i) }
    public int m1(int x) { return 10; }
}
```

m1(int) is already defined  
in Test

Test t = new Test();  
t.m1(10);

8

## Overloading

Two methods are said to be overloaded if and only if both methods having same name but different argument types.

⇒ In C language method overloading concept is not available hence we can't declare multiple methods with same name but different argument types. If there is change in argument type compulsorily we should go for new method name which increases complexity of programming.

Ex:

C

`abs (int i) ⇒ abs (10);`

`labs (long l) ⇒ labs (10L);`

`fabs (float f) ⇒ fabs (10.5f);`

⇒ But in Java we can declare multiple methods with same name but different argument types. Such type of methods are called Overloaded methods.

Ex:

Java

`abs (int i)`

`abs (long l)`

`abs (float f)`

} Overloaded methods

having Overloading concept in Java reduces Complexity of programming.

Ex:

Class Test

{ public void m1()

} { System.out.println("no-arg"); }

public void m1(int i)

{ System.out.println("int-arg"); }

public void m1(double d)

{ System.out.println("double-arg"); }

Overloaded  
methods

pr v main (String[] args)

{

Test t = new Test();

t.m1(); no-arg

t.m1(10); int-arg

t.m1(10.5); double-arg.

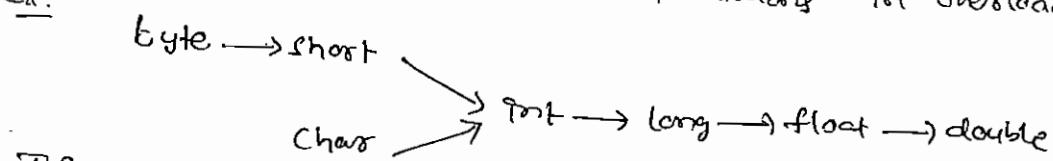
\* An Overloading method resolution always takes care by Compiler based on Reference type. Hence overloading is also considered as Compiletime Polymorphism (or) Static Polymorphism (or) early binding.

## Case 1 Automatic promotion in overloading

Session

While Resolving available then overloaded methods if exact matched method is not first it will promote argument to the next-level and check whether matched method is available or not. If matched method is available then it will be considered, and if the matched method is not available then compiler promotes argument once again to the next-level promotions. Still if the matched method is not available then we will get compiletime error.

The following are all possible promotions in overloading



This process is called automatic promotion for overloading.

Ex:

Class Test

{

```

    public void m1(int i)
    {
        System.out.println("int-arg");
    }

    public void m1(float f)
    {
        System.out.println("float-arg");
    }
  
```

over loaded methods

}

P : main(String[] args)

{ Test t = new Test();

t.m1(10); int-arg

t.m1(10.5f); float-arg

t.m1('a'); int-arg

t.m1(10.5); float-arg

t.m1(10.5);

double ↴ C-E

Can not find symbol

Symbol : method m1(double)

Location : class test

Case 2:

Ex:

Class Test

{

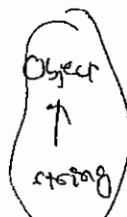
```

    public void m2(String s)
    {
        System.out.println("String version");
    }

    public void m2(Object o)
    {
        System.out.println("Object version");
    }
  
```

overloaded methods

}



{

P : main(String[] args)

{ Test t = new Test();

t.m2(new Object()); Object version  
 t.m2("durga"); String version  
 t.m2(null); String version

### Note

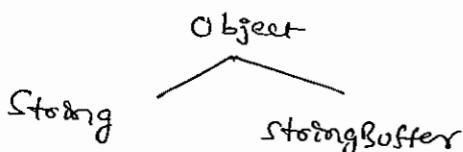
while resolving overloaded methods compiler will always give the precedence for child type argument when compared with parent type argument.

(xii)

### Case 3

```
Class Test
```

```
{  
    public void m1(String s)  
    {  
        System.out.println("String version");  
    }  
  
    public void m1(StringBuffer sb)  
    {  
        System.out.println("StringBuffer version");  
    }  
}
```



```
Program main(String[] args)
```

```
{  
    Test t = new Test();  
    t.m1("durga"); // String version  
    t.m1(new StringBuffer("durga"));  
    t.m1(null); // StringBuffer version  
}
```

C.E.: Reference to m1() is ambiguous

### Case 4

```
Class Test
```

```
{  
    public void m1(int i, float f)  
    {  
        System.out.println("int - float version");  
    }  
  
    public void m1(float f, int i)  
    {  
        System.out.println("float - int version");  
    }  
}
```

Changes in order of arguments

exact match get chosen first.

```
Program main(String[] args)
```

```
{  
    Test t = new Test();  
    t.m1(10, 10.5f); // int - float version  
    t.m1(10.5f, 10); // float - int version  
    t.m1(10, 10);  
    t.m1(10.5f, 10.5f);  
}
```

CE:  
Can not find symbol

Symbol: method m1(float, float)  
location: Class Test

CE:  
Reference to m1() is ambiguous

Case 5: class Test

```

    {
        public void m1(int x)
    }

    {
        System.out.println("General method");
    }

    public void m1(int... x)
    {
        System.out.println("Var-arg method");
    }
}

```

p.s.v.main(String[] args)

d Test t = new Test();

t.m1(); Var-arg method

t.m1(10, 20); Var-arg method

t.m1(10); General method

In General Var-arg method will get least priority i.e. if no other method matched then only Var-arg method will get the chance  
 It is exactly same as default case inside switch

Case 6:

```

class Animal
{
}

class Monkey extends Animal
{
}

class Test
{
    public void m1(Animal a)
    {
        System.out.println("Animal version");
    }

    public void m1(Monkey m)
    {
        System.out.println("monkey version");
    }
}

```

Overloaded methods

p.s.v.main(String[] args)

d Test t = new Test();

(1) Animal a = new Animal();

t.m1(a); Animal version

(2) Monkey m = new Monkey();

t.m1(m); monkey version

(3) Animal a1 = new Monkey();

t.m1(a1); Animal version

Note:

In Overloading method resolution always takes care by compiler based on reference type.  
 In overloading Runtime object won't play any role

monday

22-09-14

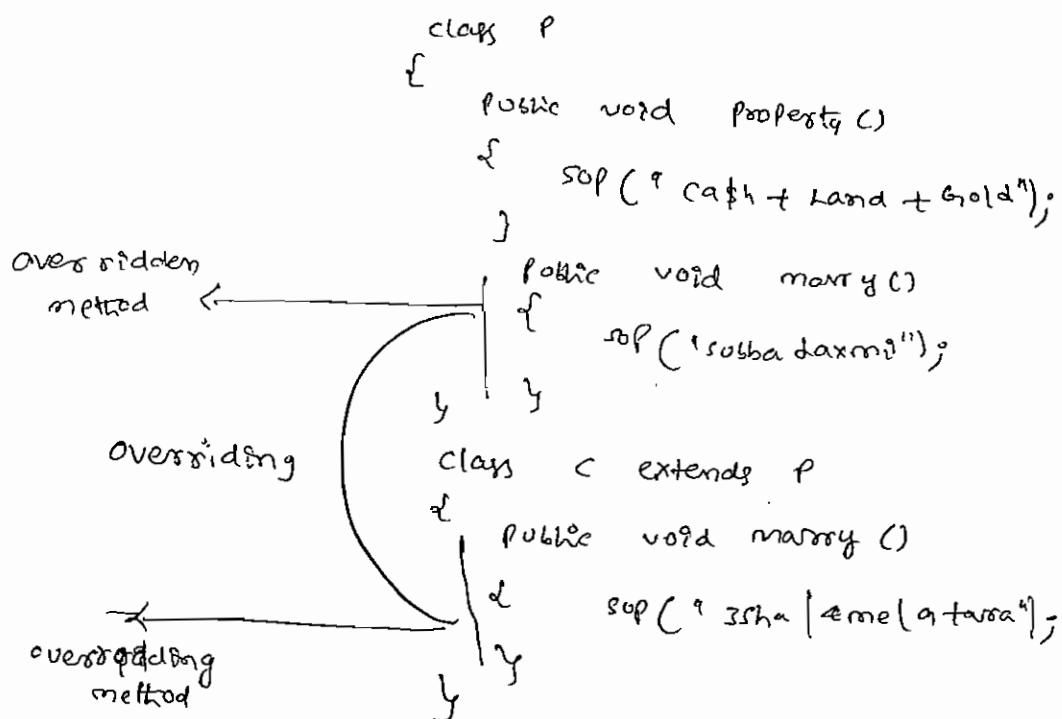
Answers

## OVERRIDING

Whatever methods parent has by default available to the child through inheritance. If child class not satisfied with parent class implementation then child is allowed to redefine that method based on its requirement this process is called overriding.

The parent class method which is overridden is called overridden method & the child class method which is overridden is called overriding method.

Ex:



Class Test  
 ↳ P s.v main(String[] args)

① P p = new P();  
 ↳ p.property(); → Parent method

② C c = new C();  
 ↳ c.property(); → Child method

③ P p1 = new C();  
 ↳ p1.property(); → Child method

{ role of  
overriders &  
sum

In overriding method resolution always takes care by JVM based on runtime object and hence overriding is also considered as runtime polymorphism or dynamic Polymorphism or late binding.

### Rules for overriding

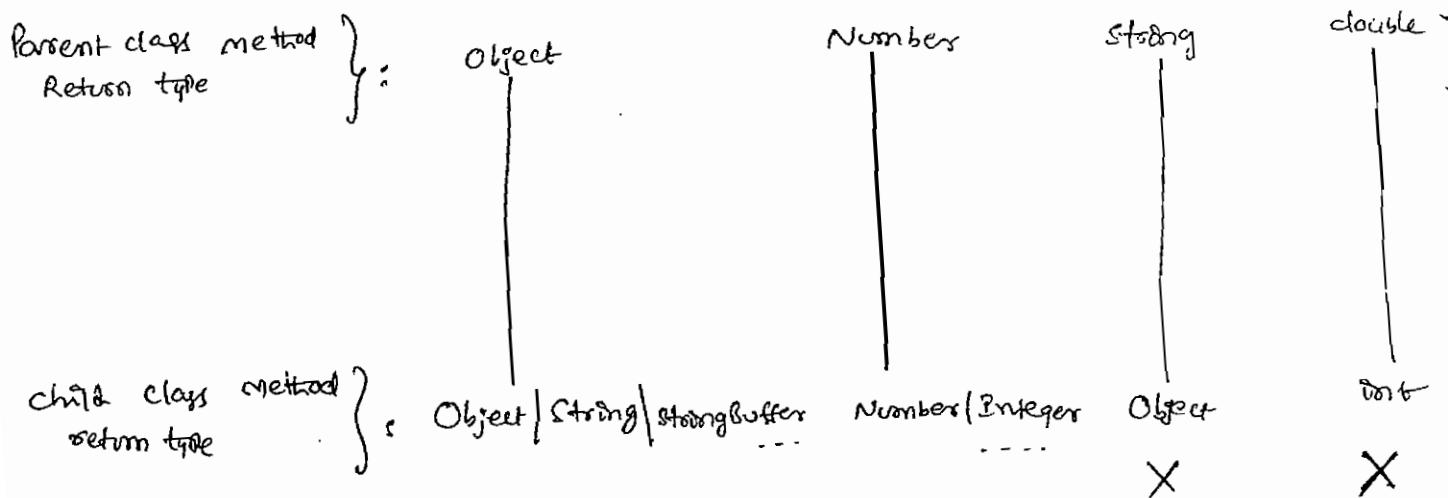
- (1) In overriding method names and argument types must be matched i.e. method signatures must be same.
- (2) In overriding return types must be same but this rule is applicable until 1.4 version only. From 1.5 version onwards we can take co-variant return types. According to this child class method return type need not be same as parent method return type. It's child type also allowed.

Ex:

```
class P
{
    public Object m1()
    {
        return null;
    }
}

class C extends P
{
    public String m1()
    {
        return null;
    }
}
```

it is invalid in 1.4 version  
but from 1.5 onwards  
it is valid

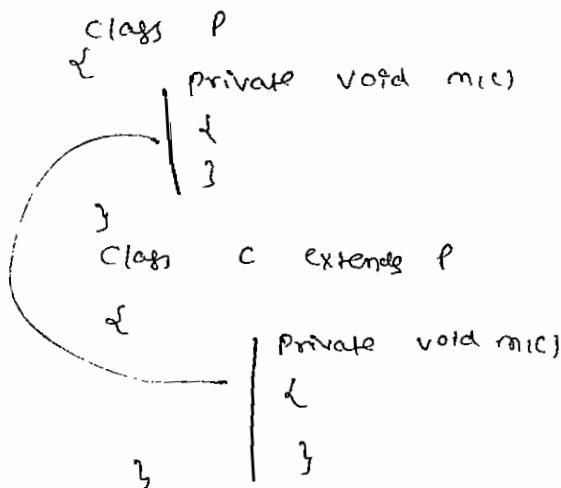


\* Co-variant return type concept applicable only for Object types but not for primitive types.

modified  
\* Parent class private methods not available to the child and hence overriding concept not applicable for private methods.

Based on our requirement we can define exactly same private method in child class it is valid but not overriding

Ex:



- \* we can't override parent class final methods in child classes  
If we are trying to override we will get compilation error

Ex: Class P

```
{ public final void m() }  
}  
}  
}  
}  
}  
}
```

C.B.  
m. in C can not override m.  
in P;  
overridden method is final

- \* Parent class abstract methods we should override in child class to provide implementation

Ex:

```
abstract Class P  
{  
    public abstract void m();  
}  
}  
}  
}
```

\* We can override non abstract method as abstract.

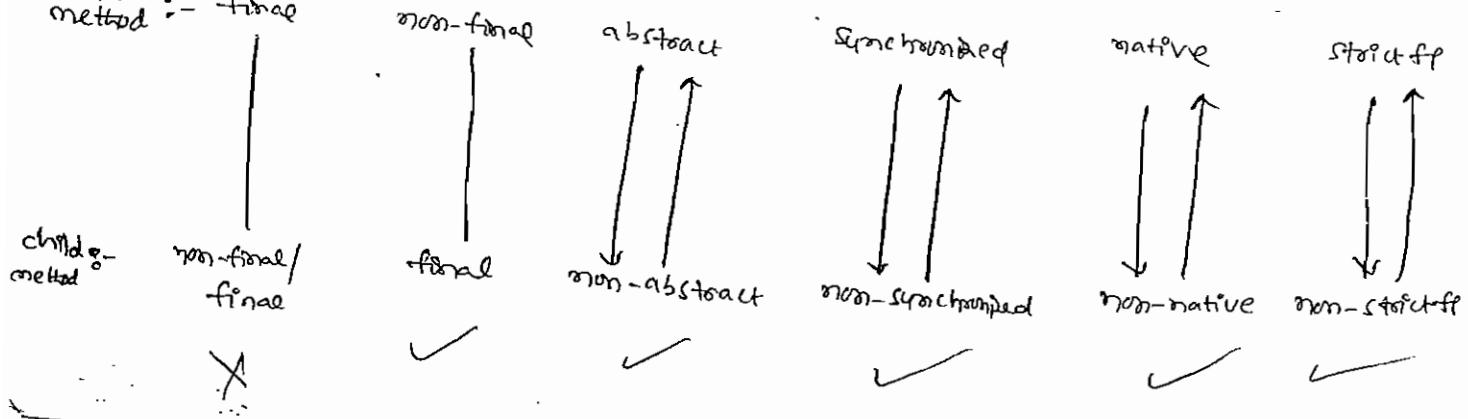
```
class P
{
    public void m1()
}

abstract class C extends P
{
    public abstract void m1();
}
```

The main advantage of this approach is we can stop the availability of parent method implementation to the next level child classes.

- \* In overriding the following modifiers won't keep any restriction
- (1) synchronized
  - (2) native
  - (3) strictfp

Parent method :- final



While overriding we can't reduce scope of access modifier but we can increase the scope.

```
class P
{
    public void m1()
}

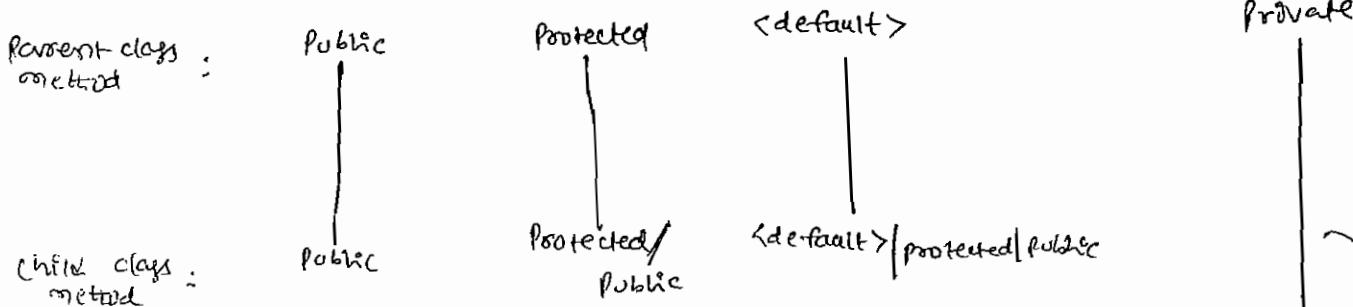
class C extends P
{
    void m1()
}
```

C.E:

m1 in C cannot override m1() in P; attempting to assign weaker access privileges; was public

## [ Private < default < Protected < Public ]

pass



23-09-14

Overriding concept is not applicable for private methods

- If child class method throws any checked exception compulsorily Parent class method should throw the same checked exception (or) it's parent otherwise we will get compiletime error but there are no restrictions for unchecked exceptions.

Ex:

```

import java.io.*;
class P {
    public void m1() throws IOException {
    }
}
class C extends P {
    public void m1() throws EOFException, InterruptedException {
    }
}
  
```

C.E: m1() in C can not override m1() in P; overridden method does not throw java.lang. InterruptedException

① P: public void m1() throws Exception

C: public void m1()

X ② P: public void m1()

C: public void m1() throws Exception

③ P: public void m1() throws Exception

C: public void m1() throws IOException

- ④ P: public void m1() throws IOException
- X C: public void m1() throws Exception
- ✓ ⑤ P: public void m1() throws IOException
- C: public void m1() throws FileNotFoundException, EOFException
- X ⑥ P: public void m1() throws IOException
- C: public void m1() throws EOFException, InterruptedException
- ✓ ⑦ P: public void m1() throws IOException
- C: public void m1() throws NullPointerException, ClassCastException

### Overriding with respect to static methods

① we can't override a static method as non static otherwise we will get compiletime error

```

Ex: class P
    {
        public static void m1()
    }
}

class C extends P
{
    public void m1()
}
  
```

*C\_Error*

m1() in C cannot override  
m1() in P; overridden  
method is static

② Similarly we can't override a non static method as static.

```

Ex: class P
    {
        public void m1()
    }
}

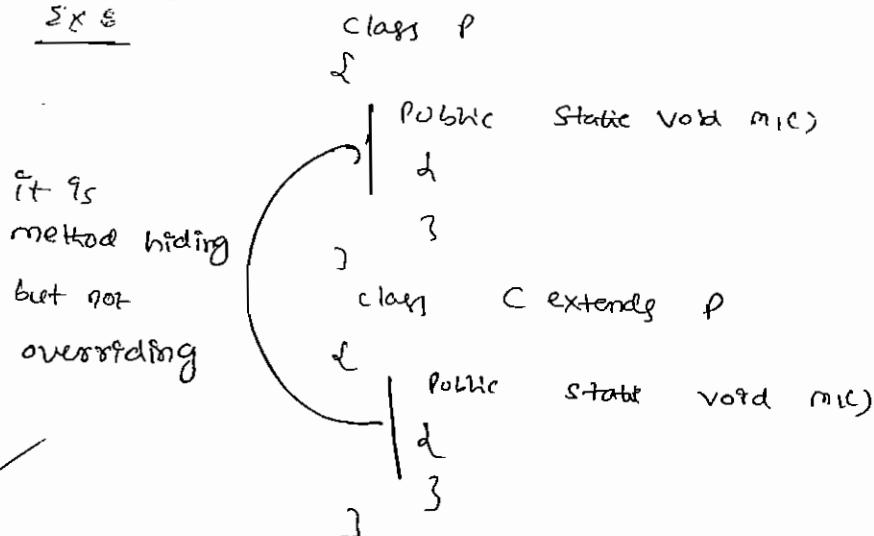
class C extends P
{
    public static void m1()
}
  
```

*C\_Error*

m1() in C cannot override m1()  
in P; overriding method  
is static

- ③ If both parent & child class methods are static then we won't get any compiletime error. It seems overriding concept applicable for static methods but it is not overriding and it is method hiding.

Ex:



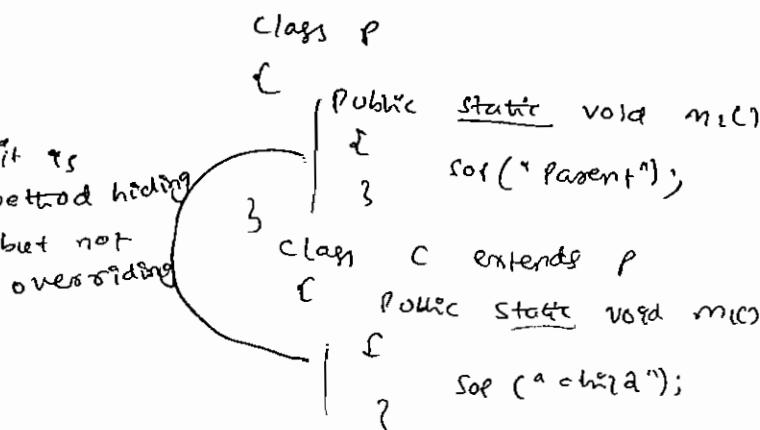
### Method Hiding

All rules of method hiding are exactly same as overriding except the following differences

#### Method hiding

- ① Both Parent & child class methods should be static
- ② Compiler is responsible for method resolution based on reference type
- ③ It is also known as Compiletime Polymorphism (or) static Polymorphism (or) early binding

#### Programs



#### overriding

- ① Both parent & child class methods should be non static
- ② JVM is always responsible for method resolution based on runtime object.
- ③ It is also known as runtime Polymorphism (or) dynamic Polymorphism (or) late binding

class Test

{

P = > main(String[] args)

{

P p = new PC();

p.m1(); → Parent

C c = new CC();

c.m1();

→ Child

P p1 = new CC();

p1.m1();

→ Child Parent

}

}

- If both Parent & child class methods are non static then it will become overriding. In this case output is

26/09/16

Parent  
child  
child

Overriding with respect to Var-arg methods

We can override var-arg method with another var-arg method only if we are trying to override with normal method then it will become overloading but not overriding.

like

class P

{

public void m1(int... x)

{

SOP("Parent");

}

class C extends P

{

public void m1(int x)

{

SOP("child");

}

class Test

{

P = > main(String[] args)

{

P p = new PC();

p.m1(10); → Parent

C c = new CC();

c.m1(10); → Child

P p1 = new CC();

p1.m1(10); → Child Parent

}

}

In the above program if we replace child method with warning methods then it will become overriding. In this case the output is

Parent  
child  
child

### OVERRIDING WITH RESPECT TO VARIABLES

Variable resolution always takes care by compiler based on reference type irrespective of whether the variable is static (or) non static (Overriding concept applicable only for methods but not for variables).

Ex: class P

```
int x=888;  
}  
class C extends P  
{  
    int x=999;  
}
```

class Test

```
P p = new P();  
p.x
```

```
SOP(p.x); → 888
```

```
C c = new C();  
SOP(c.x); → 999
```

```
P p1 = new C();  
SOP(p1.x); → 888
```

P → non static  
C → non static  
888  
999  
888

P → static  
C → non static  
888  
999  
888

P → non static  
C → non static  
888  
999  
888

P → static  
C → static  
888  
999  
888

### Differences between Overloading & overriding

Property	Overloading	overriding
① Method Names	must be same	→ must be same
② Argument Types	must be different (At least one)	→ must be same (including order)
③ method signatures	must be different	→ must be same
④ Return Types	NO Restriction	→ must be same until 1.4 version but from 1.5 onwards Co-variance return types also allowed
⑤ private, static, final methods	can be overloaded	→ can not be overridden

6. Access modifiers

No Restrictions

We can't reduce scope of Access modifier but we can increase the scope.

7. Throws clause

No Restrictions

If child class method throws any checked exception (necessarily parent class method should throw the same checked exception as its parent) but no restriction for unchecked exceptions.

8. Method Resolution

Always takes care by compiler based on reference type

Always takes care by JVM based on Runtime object.

9. It is also known as

Compiletime Polymorphism (or)  
Static Polymorphism (or) Early binding

Runtime Polymorphism (or)  
Dynamic Polymorphism (or)  
Late binding

- Notes
- In overloading we have to check only method names (must be same) and argument types (must be different). We are not required to check remaining like return types, access modifiers etc...
  - but in overriding everything we have to check like method names, argument types, return types, access modifiers, throws clause etc.

lets

Consider the following method in Parent class

Public void m1 (int x) throws IOException

In the overriding (1)

child class which over the following methods we can take

Public void m1 (int i)

Public static int m1 (long l)

Public static void m1 (int i)

Public void m1 (int i) throws exception

Public static abstract void m1 (double d);

Illegal combination of modifiers

overriding (2)

overriding (3)

overriding (4)

X (5)

CSE

## Polymerphism

one name but multiple forms is the concept of Polymerphism.

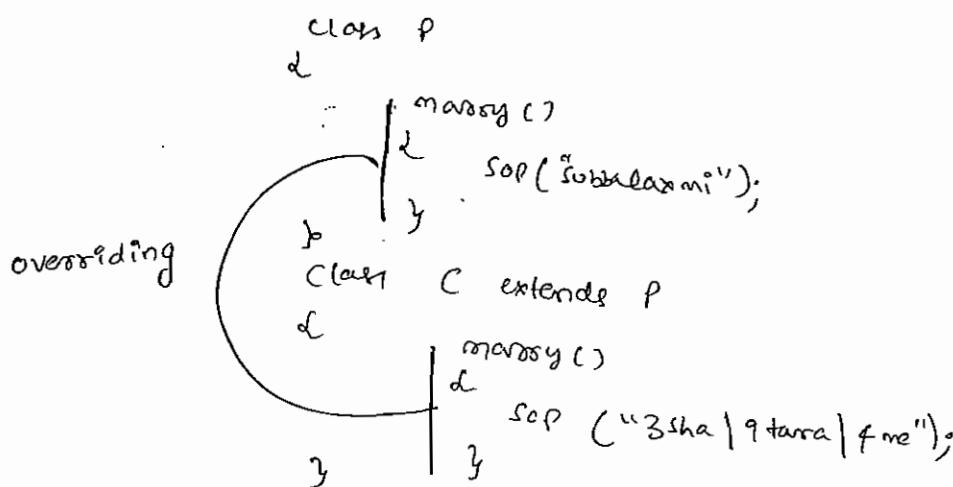
### Ex1:

method name is the same but we can apply for different types of arguments (overloading)

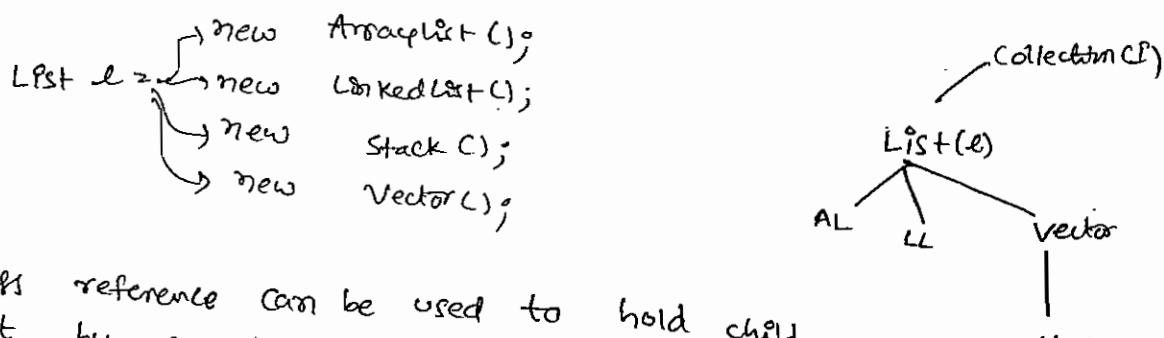
```
abs(int)
abs(long)
abs(float)
```

### Ex2:

Method signature is same but in Parent class one type of implementation & in the child class another type of implementation. (overriding)



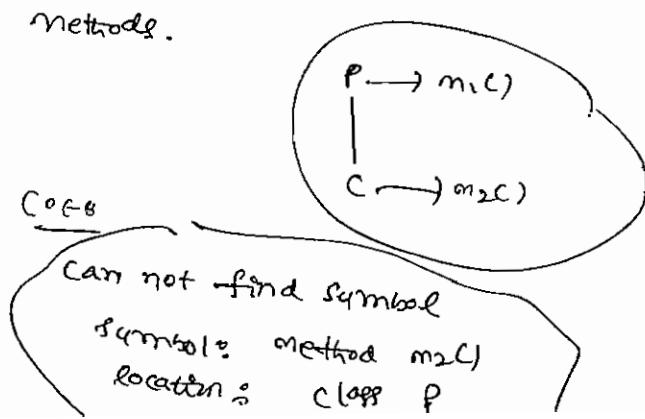
Ex3: Usage of Parent reference to hold child object is the concept of Polymerphism



Parent class reference can be used to hold child object but by using that reference we can call only the methods available in parent class and we can't call child specific methods.

but by ref

```
P p = new C();
p.m1();
p.m2(); X
```



But by using child ~~the~~ reference we can call both parent & child class methods

```
C c = new C();
c.m1(); ✓
c.m2(); ✓
```

When we should go for Parent reference to hold child object

- if we don't know exact runtime type of object then we should go for Parent reference.

for Ex: The first element present in the arraylist can be any type it may be student object (or) customer object (or) string object (or) Stringbuffer object. Hence the return type of get() method is Object which can hold any object.

Object o = l.get(0);

l → [ o | o | o | o | o ]

Heterogeneous object

C c = new C();

P p = new CC();

Eg: ArrayList l = new ArrayList();

ArrayList l = new ArrayList();

list l =  
{}  
List m1  
{  
ArrayList  
}

1) we can use this approach if we ~~know~~ know exact runtime type of object

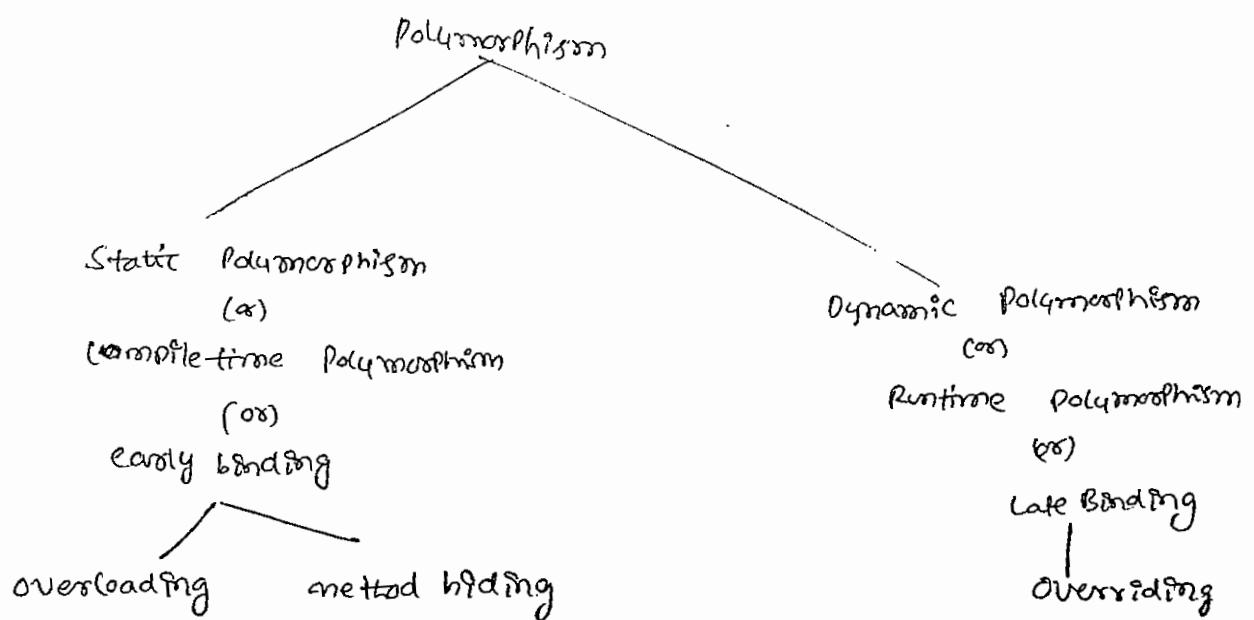
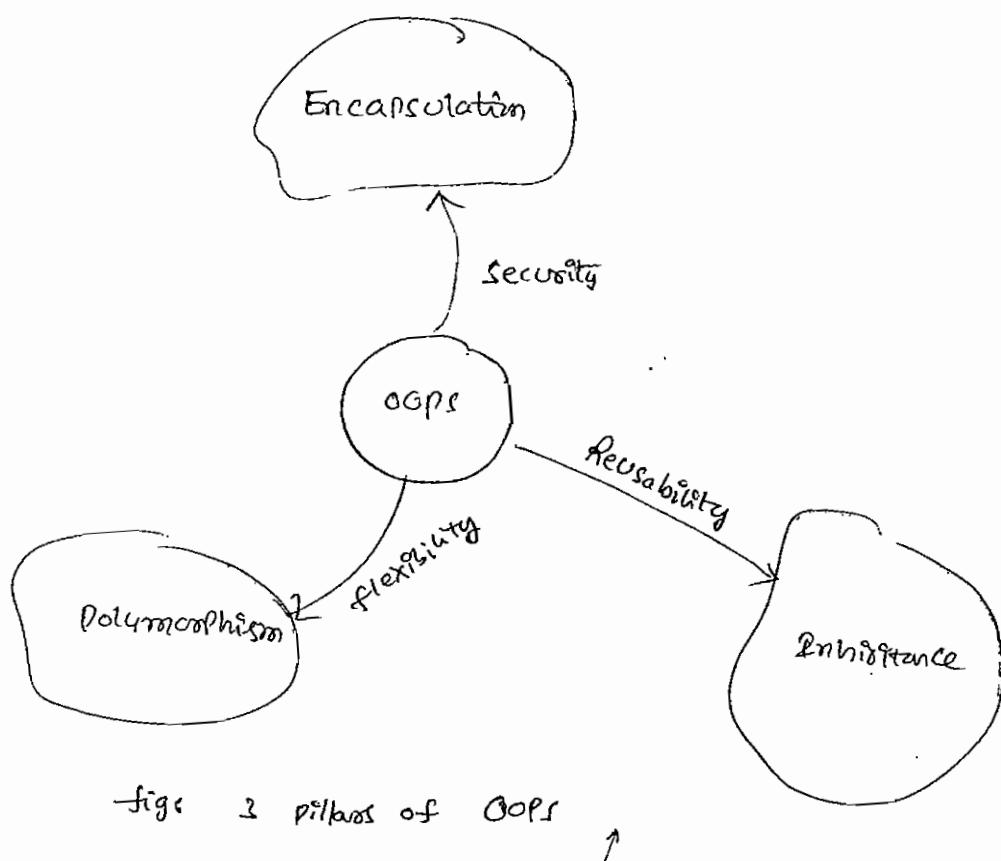
1) we can use this approach if we don't know exact runtime type of object

2) by using child reference we can call both parent class & child class methods (this is the advantage of this approach)

2) By using ~~the~~ Parent reference we can call only methods available for parent class and we can't call child specific methods (this is the disadvantage of this method)

3) we can use child reference to hold only particular child class object (this is the disadvantage of this approach)

3) we can use Parent reference to hold any child class object this is the advantage of this approach.



### Beautiful definition of Polymorphism

A BOY starts LOVE with the word FRIENDSHIP, but GIRL ends LOVE with the same word FRIENDSHIP. word is the same but attitude is different. The beautiful concept of OOPS is nothing but Polymorphism.

25-09-14

## Coupling

Coupling The degree of dependency b/w the components is called coupling.

If dependency is more then it is considered as tightly coupling and if dependency is less then it is considered as loosely coupling.

```

Ex6
class A
{
    static int i = B::j;
}
class B
{
    static int j = C::k;
}
class C
{
    static int k = D::m();
}

class D
{
    public static int m()
    {
        return 10;
    }
}

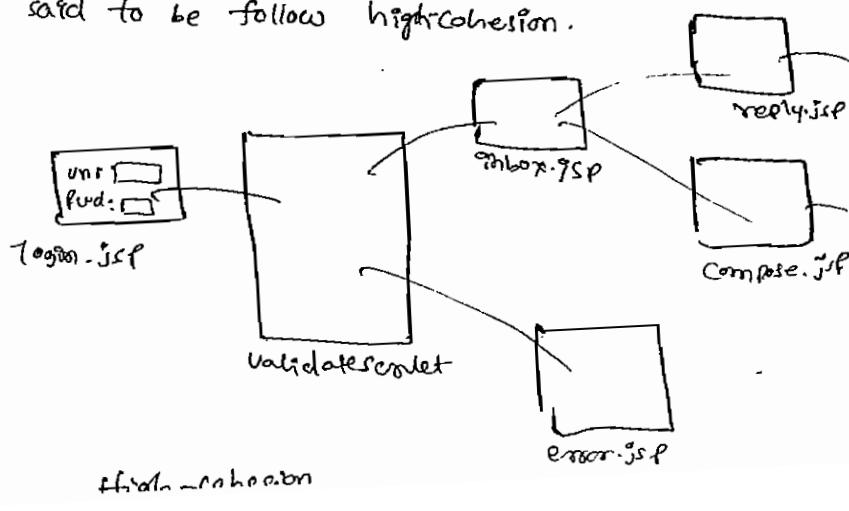
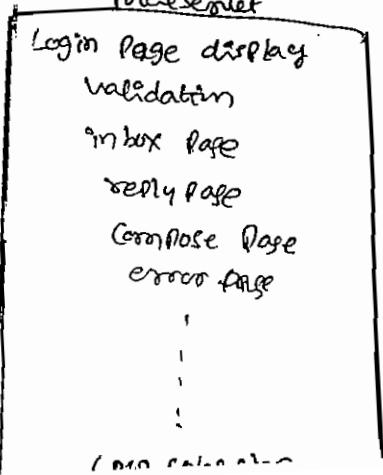
```

- The above components are said to be tightly coupled with each other because dependency b/w the components is more.
  - Tightly coupling is not a good programming practice because it has several serious disadvantages
    - ① without affecting remaining components we can't modify any component and hence enhancement will become difficult
    - ② it suppresses reusability
    - ③ it reduces maintainability of the application

Hence we have to maintain dependency b/w the components as less as possible. i.e loosely coupling is a good programming practice

## Cohesion

for every component a clear well defined functionality is required then that component is said to be follow high-cohesion.



High-cohesion is always a good programming practice because it has several advantages.

- ① without affecting remaining components we can modify any component hence enhancement will become easy
- ② it promotes reusability of the code (whenever validation is required we can reuse the same validate method without rewriting)
- ③ it improves maintainability of the application.

Note:

loosely coupling and high-cohesion are good programming practices

## Object Type-casting

We can use parent reference to hold child object

Ex:

```
Object o = new String("durga");
```

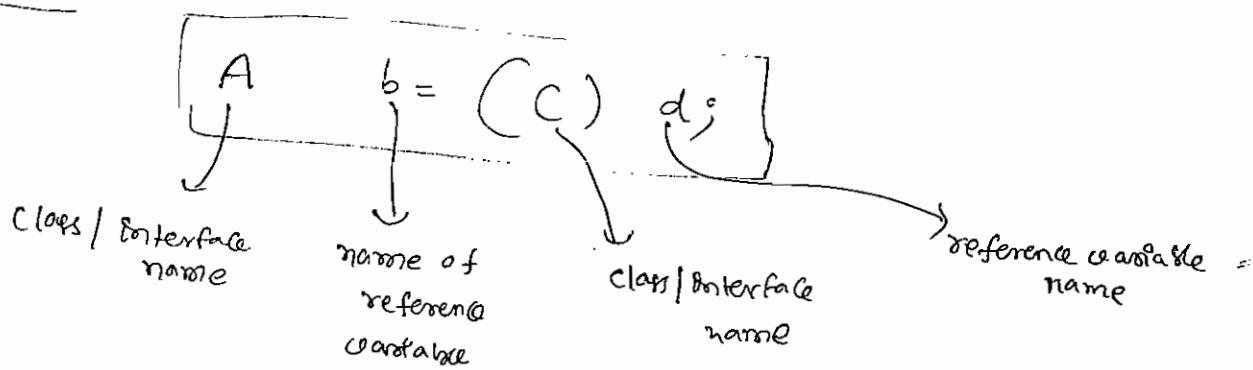
We can use

Interface reference to hold implemented class Object

Ex:

```
Runnable r = new Thread();
```

Syntax:



## Martrona (Compiletime Checking #)

The type of 'd' and 'C' must have some relation either Child to Parent (or) Parent to child (or) same type otherwise we will get compiletime error saying Inconvertible types found  
found: d type  
required: C

Ex:

```
Object o = new String("durga");
```

```
StringBuffer sb = (StringBuffer) o;
```

Valid (compiletime)

Ex 2      String s = new String ("durga");

StringBuffer sb = (StringBuffer) s;

Ex 3      Inconvertable types

found: java.lang.String

required: java.lang.StringBuffer

(... derived child)

Monto 2 (Compiletime checking 2)

'C' must be either same or derived type of 'A' otherwise we will get compiletime error saying incompatible types

Ex 4

-found: C

required: A

Object o = new String ("durga");

StringBuffer sb = (StringBuffer)o;

✓ valid (compile only)

Ex 5

Object o = new String ("durga");

StringBuffer sb = (String)o;

C-E: Incompatible types

found: java.lang.String

required: java.lang.StringBuffer

Monto 3 (Runtime checking)

Runtime Object type of 'd' must be either same or derived type of 'C' otherwise we will get runtime exception saying classcastexception

Ex 1

Object o = new String ("durga");

StringBuffer sb = (StringBuffer)o;

(String is not child of StringBuffer)

Ex 2

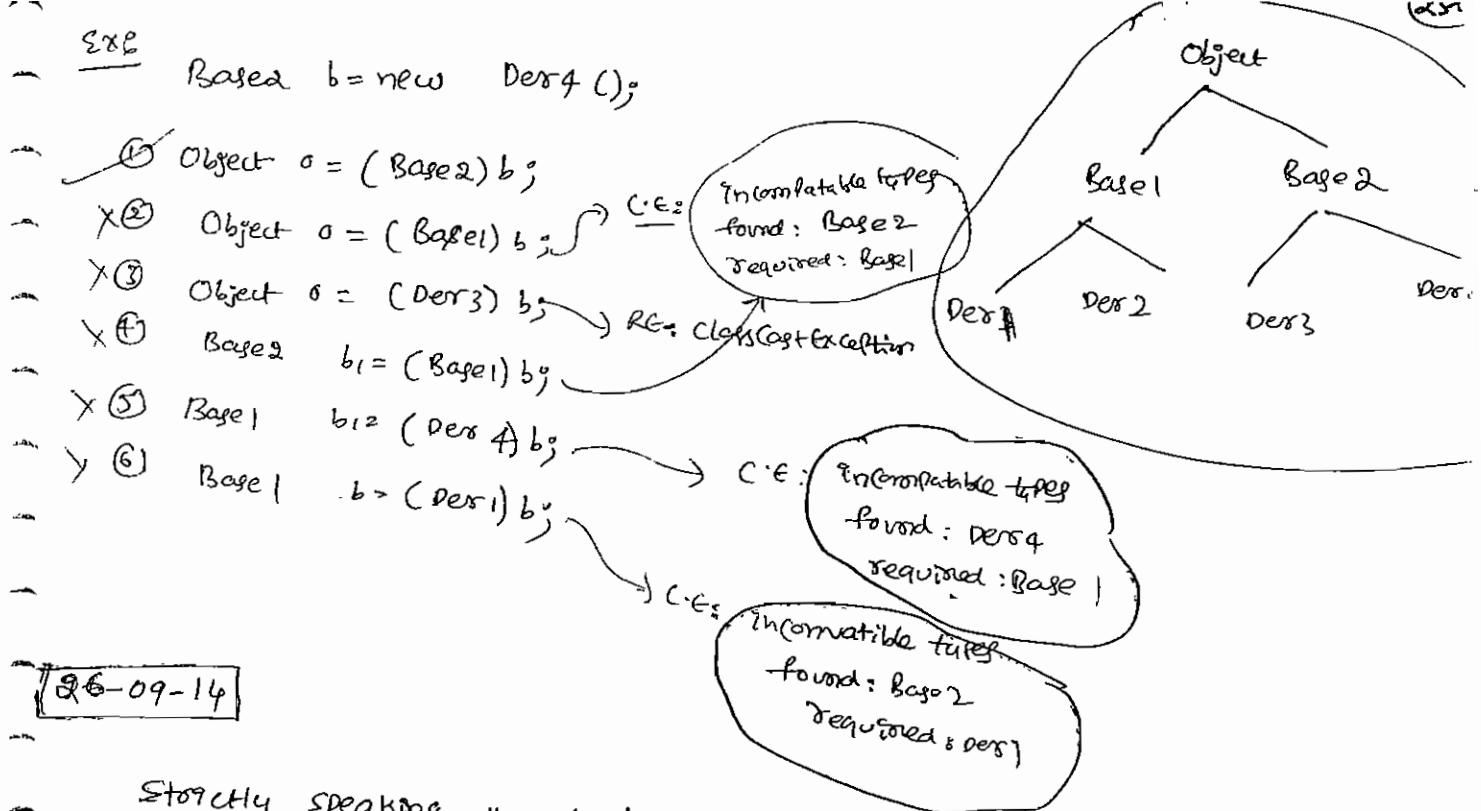
ClassCastException: java.lang.String cannot be cast to java.lang.StringBuffer

valid

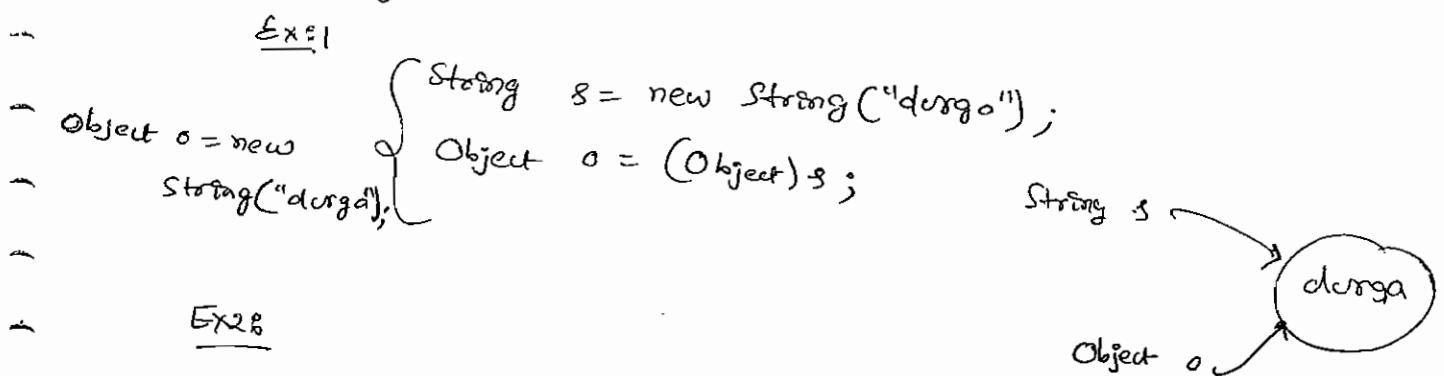
{ Object o = new String ("durga");

Object q = (String)o;

StringBuffer



Strictly speaking through type casting we are not creating any new object. for the existing object we are providing another type or reference variable. i.e we are performing typecasting but not object casting.



Ex:

```

Integer i = new Integer(10);
Number n = (Number)i;
Object o = (Object)n;
    
```

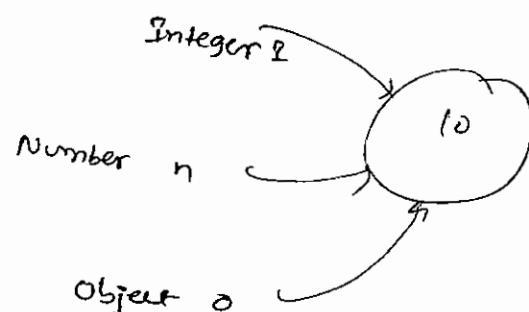
Integer i  
Number n  
Object o

```

Number n = new Integer(10);
    
```

Number n

SOP(i==n); // true  
SOP(n==o); // true



## Notes

C c=new C();  
 B b=new c(); (B)c  
 A a=new C(); (A) ((B)c)

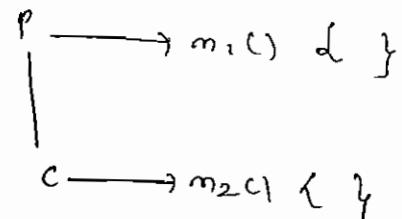


## Ex1:

(C c=new C();)

- ✓ c.m1();
- ✓ c.m2();
- ✓ ((P)c).m1();

✗ ((P)c).m2();

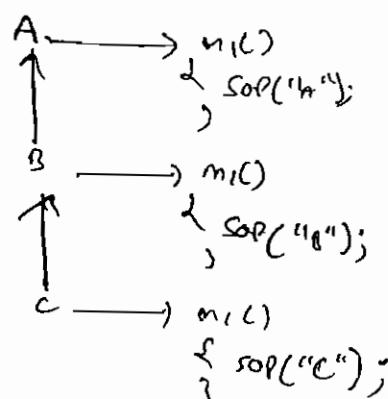


## Reason:

Parent reference can be used to hold child object but by using that reference we can't call child specific methods and we can call only the methods available in parent class.

## Ex2:

C c=new C();  
 c.m1(); → C  
 ((B)c).m1(); → C  
 ((A)((B)c)).m1(); → C

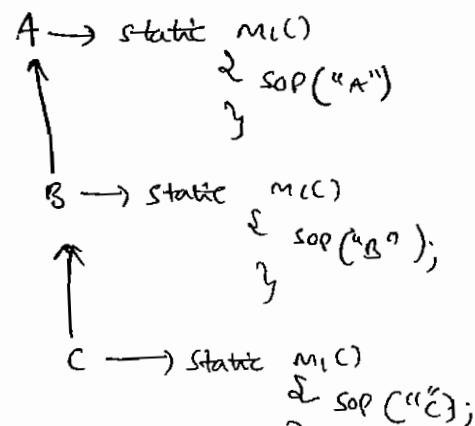


## Reason:

It is overriding and method resolution is always based on Runtime Object.

### Ex 3:

```
C c=new C();
c.m1(); -----> C
((B)c).m1(); -----> B
((A)((B)c)).m1(); -----> A
```

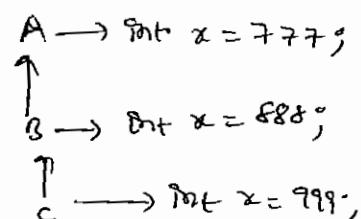


### Reason:

It is method hiding and method resolution is always based on reference type.

### Ex 4:

```
C c=new C();
sop(c.x); -----> 999
sop(((B)c).x); -----> 888
sop((A((B)c)).x); -----> 777
```



### Reason:

Variable resolution is always based on reference type but not based on runtime object.

## Static Control flow

Whenever we are executing a java class the following sequence of steps/activities will be executed as the part of static control flow.

- ① Identification of static members from top to bottom (1 to 6)
- ② Execution of static variable assignments and static blocks from top to bottom (7 to 12)
- ③ Execution of main method. (13 to 15)

### Ex:

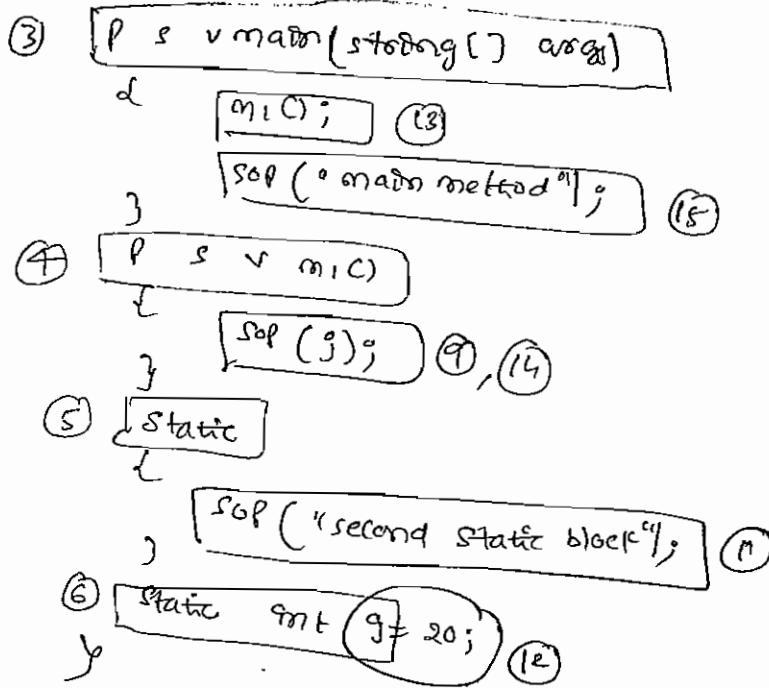
#### Class Base

```
i=0 [R1W0]
j=0 [R2W0]
i=10 [E1W]
j=20 [R2W]
```

↳

```

  ① static int i=10; ⑦
  ② static {
      m1();
      sop("First static Block"); ⑩
    }
  }
```



⑦  $\Rightarrow$  default value

Java Base  
 o  
 first static block  
 second static block  
 20  
 main method

### Read Indirectly Write only

Inside static block if we are trying to read a variable that read operation is called direct read.  
 If we are calling a method and with in that method is called trying to read a variable that read operation is called indirect read.

Ex:

```

class Test
{
    static int i=10;
    static
    {
        m();
        Sop(i); → Direct read
    }
    P s v m();
    } Sop(i); → Indirect read

```

If a variable is just identified by the JVM and original value not yet assigned then the variable is said to be in read indirectly & write only state (RIWO)

If a variable is in read indirectly write only state then that read operation so we can't perform Direct read but we can perform Indirect read.

If we are trying to read directly then we will get Compiletime error saying Illegal forward reference

l6w  
↳ all

Ex1:

```
class Test
{
    ① static int x=10;
    ② static {
        System.out.println(x);
    }
}
```

O/P: 10

R.E: NoSuchMethodError: main

class Test
{
 ① static {
 System.out.println(x);
 }
 ② static int x=10;
}

x=10 [REWO]

C.E: Illegal forward reference

class Test

```
① static {
    main();
}
}
```

② public static void main()

System.out.println(x);

```
③ static int x=10;
}
```

O/P: 0

R.E: NoSuchMethodError:  
main

### Static block

Static blocks will be executed at the time of class loading hence at the time of class loading if we want to perform any activity we have to define that inside static block.

Ex1: At the time of java class loading the corresponding native libraries should be loaded hence we have to define this activity inside static block.

```
class Test
{
    static {
        System.loadLibrary("native library path");
    }
}
```

Ex2: After loading every database Driver class we have to register Driver with DriverManager but inside Database Driver class there is static block to perform this activity and we are not responsible to register explicitly.

```

class Drivers
{
    static
    {
        Register this Driver
        with Driver Manager
    }
}

```

Note 8 With in a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

(Q1) Without writing main() method is it possible to print some statements to the console?

A: Yes, By using static block.

Ex 8

```

class Test
{
    static
    {
        System.out.println("Hello 2 can print");
        System.exit(0);
    }
}

```

O/P: Hello 2 can print.

(Q2) Without writing main() method & static block is it possible to print some statements to the console?

A: Yes, ofcourse there are multiple ways

Ex 9

```

class Test
{
    static int x=m();
    public static int m()
    {
        System.out.println("Hello 1 can print");
        System.exit(0);
        return 10;
    }
}

```

(2) class Test

```

static Test t=new Test();
{
    System.out.println("Hello 2 can print");
    System.exit(0);
}

```

(3)

```

class Test
{
    static Test t=new Test();
    Test()
    {
        System.out.println("Hello 3 can print");
        System.exit(0);
    }
}

```

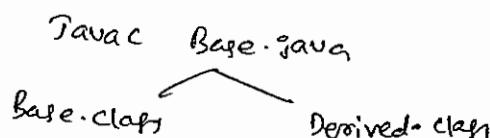
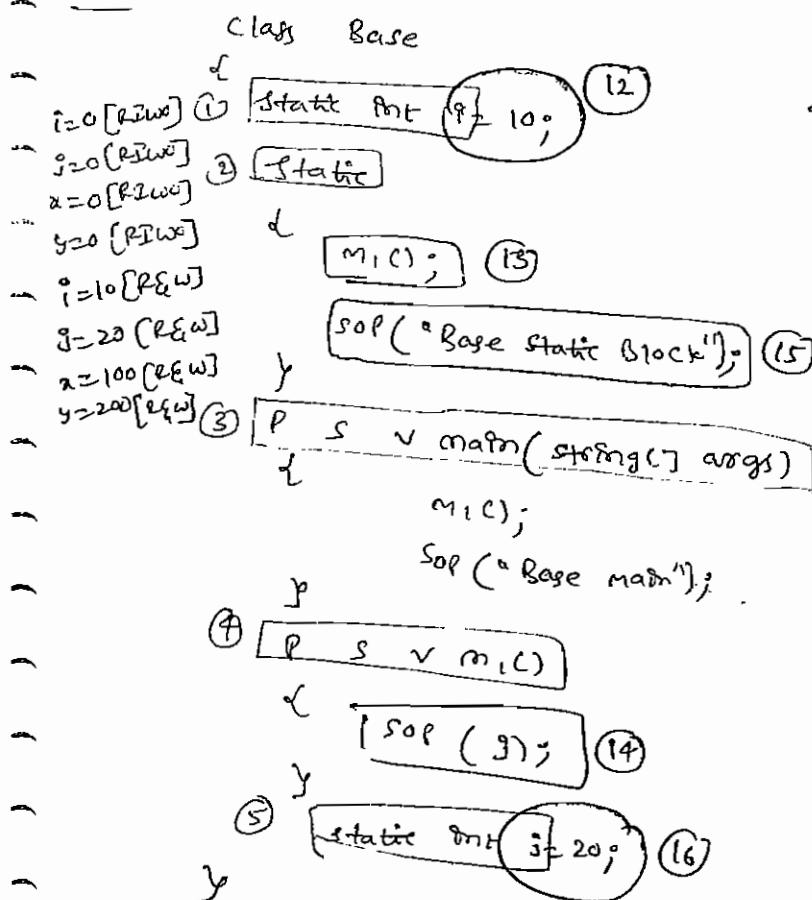
NOTE from 1.7 version onwards `main()` method is mandatory to start program execution hence from 1.7 version onwards without writing `main()` method it is impossible to print some statements to the console.

### Static control flow in Parent to child relationship

Whenever we are executing child class the following sequence of events will be executed automatically as the part of static control flow.

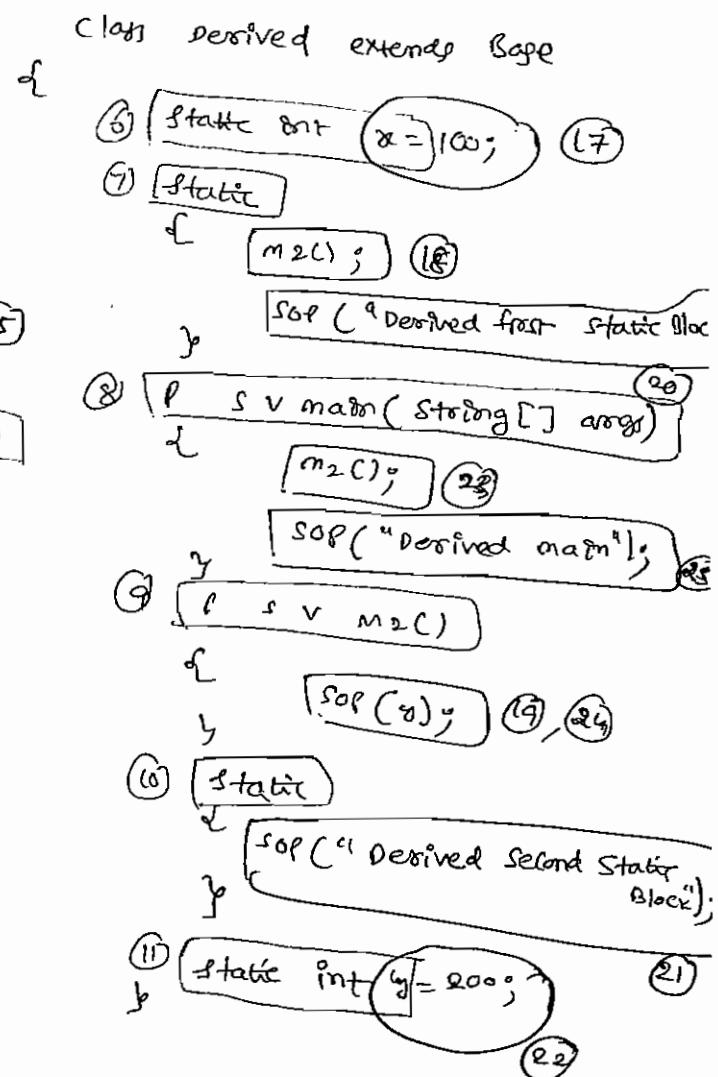
- (1) Identification of static members from parent to child
- (2) Execution of static variable assignments & static blocks from parent to child (1 to 11)
- (3) Execution of only child class main() method (12 to 25)

Ex:



Q/P: Java Derived

O  
Base static Block  
O  
Derived First static Block  
Derived Second static Block  
200  
Derived main



Java Base

Q/P:

O  
Base static Block  
20  
Base main

Note: Whenever we are loading child class automatically parent class will be loaded.  
 but whenever we are loading parent class child class won't be loaded (Because Parent class members by default available to the child class, where as child class members by default can't be available to the parent)

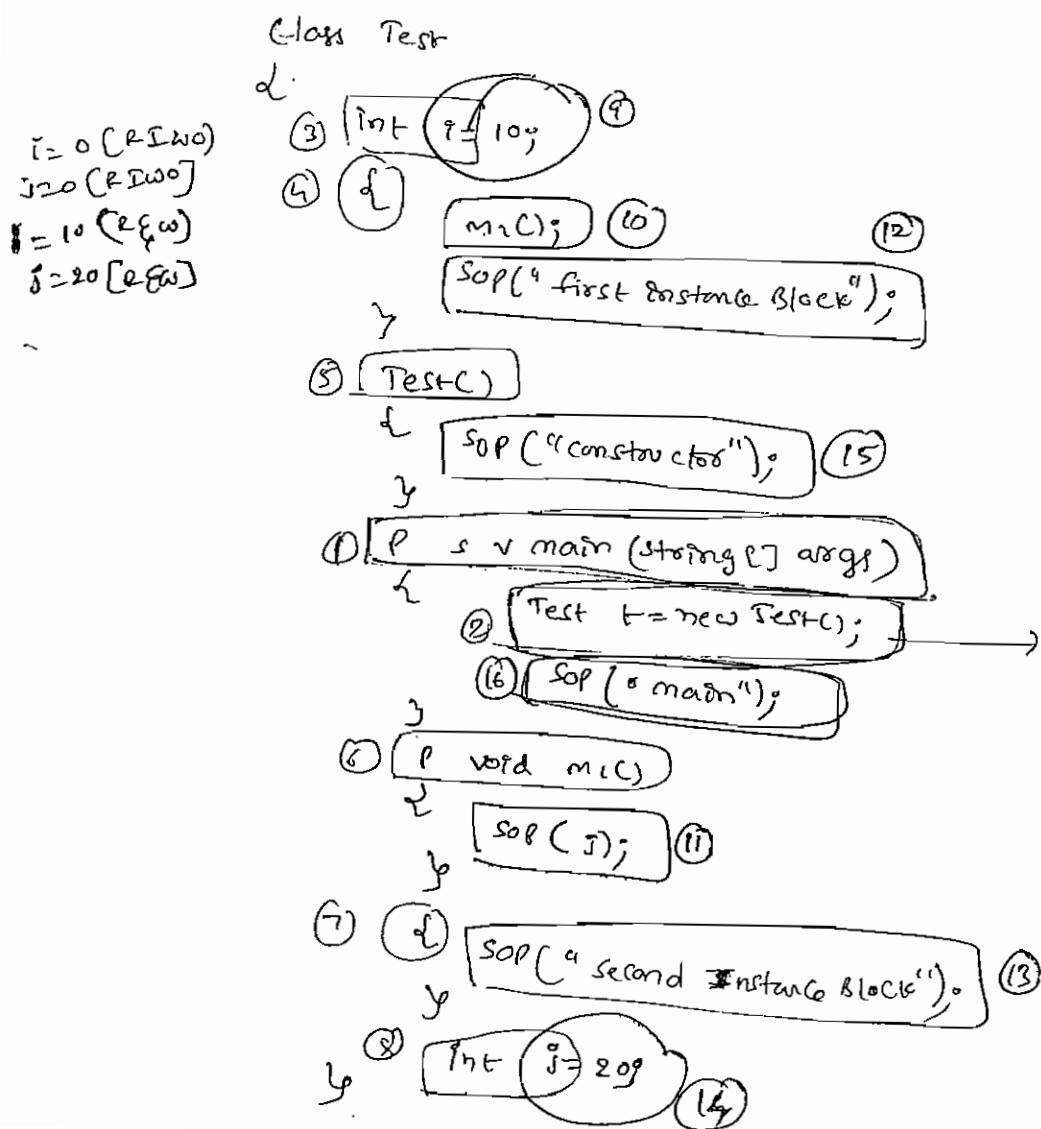
## Instance Control flow

Whenever we are executing a java class first static control flow will be executed.

In the static control flow if we are creating an object the following sequence of events will be executed as the part of instance control flow.

- ① Identification of instance members from top to bottom (3 to 8)
- ② Execution of instance variable assignments and instance blocks from top to bottom. (9 to 14)
- ③ Execution of constructor. (15)

## Ex:



If we comment line 1 then the output is main

Note:

- (1) Static control-flow is one-time activity which will be performed at the time of class loading.  
but ~~static~~ instance control flow is not one-time activity and it will be performed for every object creation.
- (2) Object creation is the most costly operation if there is no specific requirement then it is not recommended to create object.

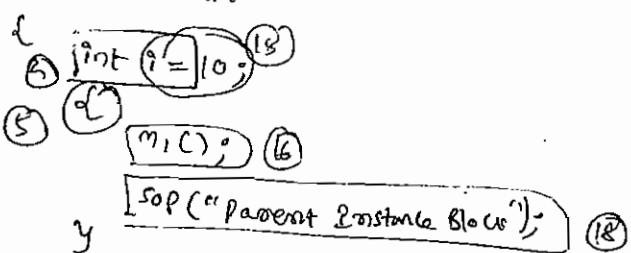
27/09/2014

### instance control flow in parent to child relationship

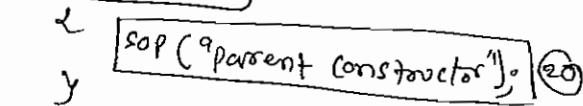
Whenever we are creating child class object the following sequence of events will be performed automatically as the part of instance control flow.

- (1) Identification of instance members from parent to child [4 to 14]
- (2) Execution of instance assignable assignments & instance blocks
- (3) only in parent class (15 to 19)
- (4) execution of parent constructor (20)
- (5) execution of instance assignable assignments & instance blocks in child class (21 to 26)
- (6) execution of child constructor (27)

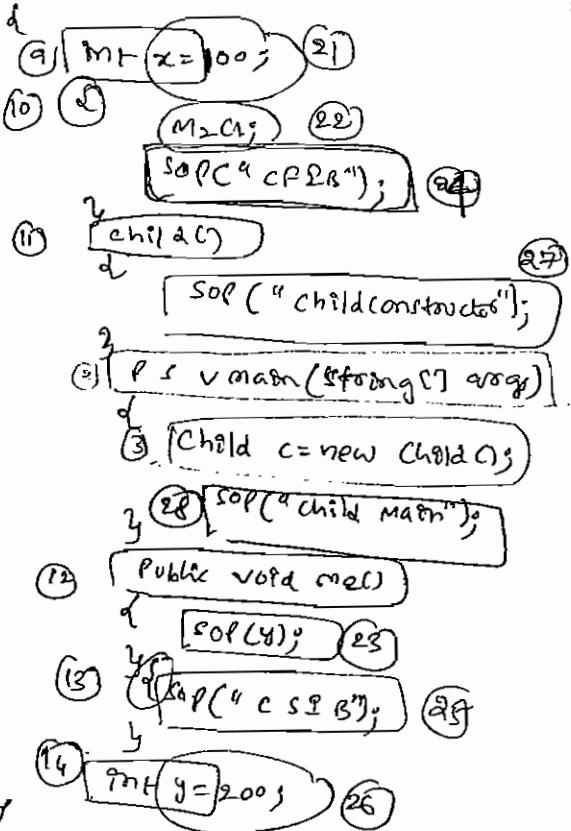
class Parent



(5) Parent C()



class Child extends Parent



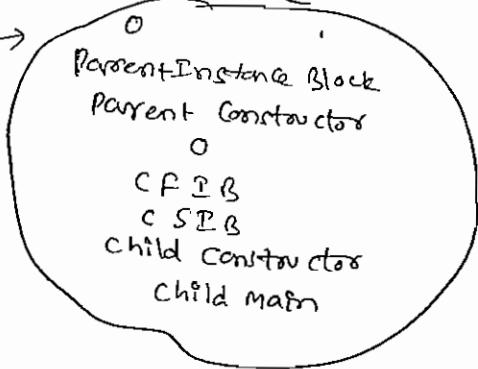
O/P:

### Java's Parent-gang



### Java's Child

O/P →



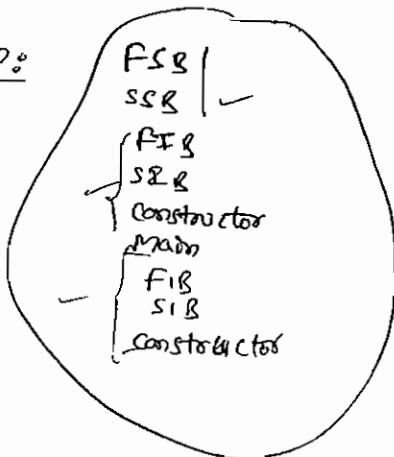
Exe

### Class Test

```

class Test {
    static {
        System.out.println("P(B)");
    }
    public void f() {
        System.out.println("FSB");
    }
    public void g() {
        System.out.println("constructor");
    }
    public void h() {
        System.out.println("toString() overr");
        Test t1 = new Test();
        System.out.println("main");
        Test t2 = new Test();
    }
    static {
        System.out.println("SSB");
    }
}
  
```

O/P:



Exe

### Public class Initialization

① [private static String m1(String msg)]

```

    {
        System.out.println(msg);
        return msg;
    }
  
```

② [public Initialization()

```

    {
        m = m1("1");
        m = m1("2");
    }
  
```

③ [System.out.println(m = m1("3"));

④ [System.out.println("toString() overr");

⑤ [Object o = new Initialization();

O/P = 2

3

1

m = null

2,

Ex 38

(Ans)

### Public class Initialization

```

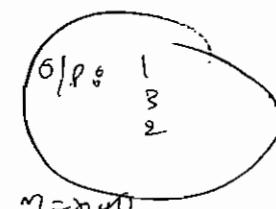
    {
        private static String m1(String msg)
        {
            System.out.println(msg);
            return msg;
        }

        static String m = m1("1");
        {
            m = m1("2");
        }

        static
        {
            m = m1("3");
        }

        public static void main(String[] args)
        {
            Object obj = new Initialization2();
        }
    }

```



### Notes

from static area we can't access instance members directly because while executing static area JVM may not identify instance members.

Ex:

```

class Test
{
    int x=10;
}

public static void main(String[] args)
{
    System.out.println(x); // Error
}

```

non-static variable x  
can not be referenced  
from a static context.

```

Test t=new Test();
System.out.println(t.x);

```

~~Ques~~  
~~Ans~~

In how many ways we can create an object in java  
(Ans)

In how many ways we can get object in Java.

① By using new operator

```
Test t = new Test();
```

② By using newInstance() method

```
Test t=(Test)Class.forName("Test").newInstance();
```

② By using factory method

Runtime r = Runtime.getRuntime();

Dateformat df = DateFormat.getInstance();

④ By using clone() method

Test t<sub>1</sub> = new Test();

Test t<sub>2</sub> = (Test)t<sub>1</sub>.clone();

⑤ By using Deserialization

fileInputStream fis = new fileInputStream("abc.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

Dog d<sub>2</sub> = (Dog)ois.readObject();

## Constructor

Once we creates an object compulsorily we should perform initialization  
then only the object is in a position to respond properly.

- whenever we are creating an object some piece of the code will be executed automatically to perform initialization of the object. this piece of the code is nothing but constructor.
- Hence the main purpose of constructor is to perform initialization of an object.

Ex:

Class Student

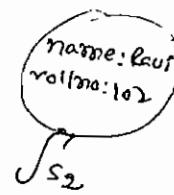
```
String name;
int rollno;
Student (String name, int rollno)
{
    this.name = name;
    this.rollno = rollno;
}
```

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

```
{ Student s1 = new Student ("durga", 101);
```

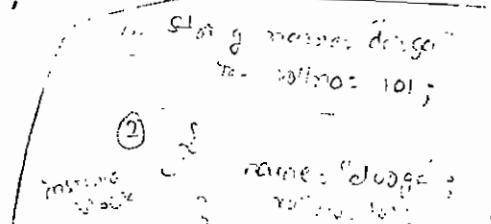
```
Student s2 = new Student ("Ravi", 102);
```

```
}
```



## Notes

The main purpose of constructor is to perform initialization of an object but not to create object.



## Difference b/w Constructor & Instance Block

The main purpose of Constructor is to perform initialization of an object  
But often initialization if we want to perform any activity  
for every object creation then we should go for instance block.  
(like updating one entry in the database for every object creation (or)  
incrementing count value for every object creation etc...)

- Both constructor & instance block have their own different purposes and replacing one concept with another concept may not work always.
- Both constructor & instance block will be executed for every object creation but instance block first followed by constructor.

Demo program, to print no. of objects created for a class.

```
class Test
{
    static int count=0;
    {
        count++;
    }
    Test()
    {
    }
    Test(int i)
    {
    }
    Test(double d)
    {
    }
}
```

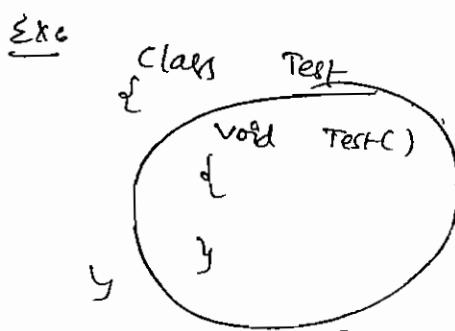
Q 5 v main(String[] args)

```
t Test t1=new Test();
Test t2= new Test(10);
Test t3= new Test(10.5);
System.out.println("The no. of objects created : "+count);
```

! Test()
{ count+=1;
}
Test(int i)
{ count+=1;
}
Test(double d)
{ count+=1;
}

### Rules of writing Constructors

- name of the class and name of the constructor must be matched.
- return type concept not applicable for constructor even void also
- By mistake if we are trying to declare return type for the constructor then we won't get any compilation error because compiler treats it as a method.



It is a method but not constructor.

hence it is legal (valid) (But stupid to have a method who's name is exactly same as class name)

Ex 6 class Test

```
{ void Test()
{ System.out.println("method but not constructor");
} public static void main(String[] args)
{
    Test t = new Test();
    t.Test();
}
```

⑦ The only applicable modifiers for constructors are

- (1) Public
- (2) Private
- (3) Protected
- (4) Default.

If we are trying to use any other modifier we will get compilation error.

Ex 6 class Test

```
{ static Test()
```

Get modifier static not allowed here.

3 ?

### default constructor

- 1) Compiler is responsible to generate default constructor (but not JVM)
- 2) If we are not writing any constructor then only compiler will generate default constructor. i.e. if we are writing atleast one constructor then compiler won't generate default constructor.
- 3) Hence every class in java can contain constructor. It may be default constructor generated by compiler or customized constructor explicitly provided by programmer. but not both simultaneously.

### prototype of default constructor

- ① It is always no-arg constructor
- ② The access modifier of default constructor is exactly same as ~~any~~ access modifier of class (This rule is applicable only for public & default)
- ③ It contains only one line

```
super();
```

It is a no argument call to super class constructor

```
class Test
{
    Test()
    {
        super();
    }
}
```

## Programmer's Code

```
class Test
{
}
```

```
public class Test
{
}
```

```
public class Test
{
    void Test()
    {
    }
}
```

```
class Test
{
    Test()
}
}
```

```
class Test
{
    Test(int i)
    {
        super();
    }
}
```

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
    }
}
```

Note

- The first line inside every constructor should be either super() or this(). If we are not writing anything then compiler will always place super().

## Compiler generated code

```
class Test
{
    Test()
    {
        super();
    }
}
```

```
public class Test
{
    public Test()
    {
        super();
    }
}
```

```
public class Test
{
    public Test()
    {
        super();
    }

    void Test()
    {
    }
}
```

```
class Test
{
    Test()
    {
        super();
    }
}
```

```
class Test
{
    Test(int i)
    {
        super();
    }
}
```

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
        super();
    }
}
```

(27)

### Case 1:

We can take `super()` or `this()` only in first line of constructor. If we are trying to take any where else we will get compiletime error.

#### Ex:

```
Class Test
{
    Test()
    {
        super("Constructor");
    }
}
```

#### C.E:

Call to super must be first statement in constructor.

### Case 2:

With in the constructor we can take either `super()` or `this()` not both simultaneously.

#### Ex:

```
Class Test
{
    Test()
    {
        super();
        this();
    }
}
```

#### C.E:

Call to this must be first statement in constructor.

### Case 3:

We can use `super()` or `this()` only inside the constructor. If we are trying to use outside of constructor we will get compiletime error.

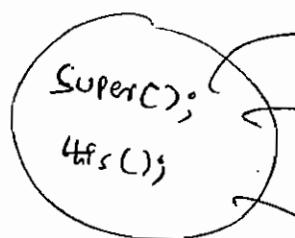
#### Ex:

```
Class Test
{
    public void m1() // method
    {
        super();
        System.out.println("Hello");
    }
}
```

#### C.E:

Call to super must be first statement in constructor.

i.e. we can call a constructor directly from another constructor only.



we can use only in constructor

only in first line

only one but not both simultaneously

super(), this()

super, this

(a)

- (1) These are constructor calls to call super class and current class constructors
- (2) We can use only in constructors as first line
- (3) we can use only once in a constructor

- (1) These are keywords to refer super class & current class instance members
- (2) we can use anywhere except static area.
- (3) we can use any number of times.

Ex:

```
Class Test
{
    public void main(String[] args)
    {
        super(super.hashCode());
    }
}
```

Ques: Non-static variable super  
cannot be referenced from a static context.

## Overloaded Constructors

With in a class we can declare multiple constructors and all these constructors having same name but different type of arguments hence ~~overload~~ all these constructors are considered as overloaded constructors. Hence overloading concept applicable for constructors

Ex:

```
Class Test
{
    Test()
    {
        this(10);
        super("no-arg");
    }

    Test(int i)
    {
        this(10-i);
        super("int-arg");
    }

    Test(double d)
    {
        super("double-arg");
    }
}
```

overloaded  
constructors

```
public void main(String[] args)
{
    Test t1 = new Test();
    Test t2 = new Test(10);
    Test t3 = new Test(10.5);
    Test t4 = new Test(10L);
}
```

double-arg  
int-arg  
no-arg

double-arg  
int-arg

double-arg  
double-arg

double-arg

automatic promotion  
comes here  
long → float  
double

- \*.) for Constructors inheritance and overriding concepts are not applicable.  
but overloading concept is applicable.
  - \*.) Every class in java including abstract class can contain constructors  
but interface can not contain constructors.

```
Class Test  
{  
    Test(c)  
}  
]  
✓
```

```
abstract class Test  
{  
    Test()  
}  
}
```

```
interface Test
{
    Test()
}

I
```

Cafelg

Recursive method call is a runtime exception saying  
StackOverflowError

(∴ every variable in interface  
is static → constructor  
purpose is to initialize  
instance variables in  
instance variable in  
interface?

but in our program if there is a chance of recursive constructor invocation then the code won't compile and we will get compiletime error.

Ex:

class Test  
 {  
 public:  
 int s = v + m1(c);  
 int m2(c);  
 };  
  
 class Test2  
 {  
 public:  
 int s = v + m2(c);  
 int m1(c);  
 };  
  
 class Test3  
 {  
 public:  
 int s = v + min (String("args"), args);  
 int m1(c);  
 };  
  
 . . .  
 . . .

```

class Test
{
    Test()
    {
        System.out.println("This(10);");
    }
    Test (int i)
    {
        System.out.println(i);
    }
    void main (String [] args)
    {
        System.out.println("Hello");
    }
}

```

C++  
Recursive Constructor Invocation

### Case 2:

```

class P
{
    P()
    {
        super();
    }
}
class C extends P
{
    C()
    {
        super();
    }
}

```



### class P

```

{
    P()
    {
        super();
    }
}

```

class C extends P

```

{
    C()
    {
        super();
    }
}

```



### class P

```

{
    P()
    {
        super();
    }
}

```

class C extends P

```

{
    C()
    {
        super();
    }
}

```



Ex: Can not find symbol  
symbol: constructor P()  
location: class P

#### \* Note:

(1) If Parent class contains any argument constructor then while writing child classes with respect to constructors we have to take special care.

(2) Whenever we are writing any argument constructor it is highly recommended to write no-arg constructor also.

### Case 3:

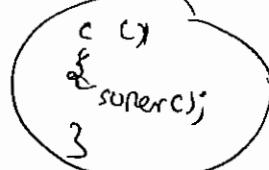
#### class P

```

{
    P() throws IOException
}

```

class C extends P



C.Es unreported exception

Java: No. IOException

In default constructor

### class P

```

{
    P() throws IOException
}

```

class C extends P

```

{
    C() throws IOException | InterruptedException | ThreadDeath
}

```

```

{
    Super();
}

```

}

m1()

m2()

m2() throws

IOException

{

=

Caller m1() should handle the checked exception by try { } catch (IOException) or throw-able built-in constructor we don't use try { } catch (Exception) because try { } catch (Exception) first statement should be super();

Note  
If parent class constructor throws any checked exception compulsorily child class constructor should throw the same checked exception or its parent. otherwise the code won't compile.

Ques 6

which of the following is valid?

- (1) The main purpose of constructor is to create an object X
- (2) The main purpose of constructor is to perform initialization of an object ✓
- (3) The name of the constructor need not be same as class name X
- (4) Return type concept applicable for constructors but only void X
- (5) We can apply any modifier for constructors. X
- (6) default constructor generated by JVM X
- (7) Compiler is responsible to generate default constructor ✓
- (8) Compiler will always generates default constructor X  
if we are not writing no-arg constructor then compiler will generate default constructor X
- (9) Every no-arg constructor is always default constructor X
- (10) default constructor is always no-arg constructor. ✓  
if we are not writing any thing then compiler will generate this () X
- (11) for constructors both overloading & overriding concepts are applicable X
- (12) only concrete classes can contain constructor but abstract class cannot X
- (13) interface can contain constructor but abstract class cannot X
- (14) recursive constructor invocation is a runtime exception X  
Exception then compulsory child class constructor should throw the same checked exception or it's parent child X  
(It is a C.E)

## Singleton classes

→ for any Java class if we are allowed to create only one object such type of class is called singleton class

Ex: Runtime  
BusinessDelegate  
ServiceLocator etc...

Advantage of Singleton class:

- ① if several people have same requirement then it is not recommended to create a separate object for every requirement. we have to create only one object and we can reuse same object for every similar requirement so that performance & memory will be improved.
- This is the central idea of singleton classes.

Ex:

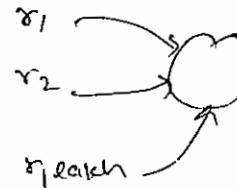
Runtime  $r_1 = \text{Runtime.getRuntime}();$

Runtime  $r_2 = \text{Runtime.getRuntime}();$

||

||

Runtime  $r_{1\text{ lakh}} = \text{Runtime.getRuntime}();$



How to create our own singleton classes

We can create our own singleton classes for this we have to use private constructor and private static variable and public factory method

Approach!

Class Test

```
Private static Test t = new Test();
Private Test()
{
}
Public static Test getTest()
{
    return t;
}
```

Note: Runtime class is internally implemented by using this approach.

```
Test t1 = Test.getTest();
Test t2 = Test.getTest();
Test t1 lakh = Test.getTest();
```

## Approach

Class Test

```

    {
        private static Test t=null;
        private Test()
        {
        }

        public static Test getTest()
        {
            if(t==null)
            {
                t=new Test();
            }
            return t;
        }
    }

```

```

Test t1=Test.getTest();
Test t2=Test.getTest();
|
Test t1 which=Test.getTest();

```

At any point of time for Test class we can create only one object  
hence Test class is singleton class

Class is not final but we are not allowed to create child classes  
How it is possible?

By declaring every constructor as private we can restrict child classes

Ex: class P  
 {
 private P()
 {
 }
 }

for the above class it is impossible to create child class

{  
 < Concentrated  
 < Private  
 < CC  
 < SuperCC  
 } X

# MultiThreading

- (1) Introduction
- (2) The ways to define a thread
  - 1) By extending Thread class
  - 2) By implementing Runnable Interface
- (3) Getting & setting Name of Thread
- \* (4) Thread priorities
- (5) The methods to prevent Thread execution
  - ① yield()
  - ② join()
  - ③ sleep()
- (6) Synchronization
- (7) InterThread Communication
- (8) DeadLock
- (9) Daemon Threads
- (10) MultiThreading Enhancements

## Introduction

### Multitasking

Executing several tasks simultaneously is the concept of multitasking. There are 2 types of multitasking.

- ① Process based multitasking
- ② Thread based multitasking

### Process based multitasking

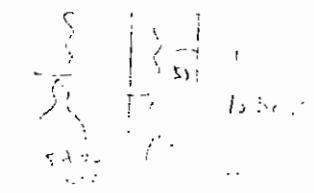
Executing several tasks simultaneously where each task is a separate independent program (~~process~~ process) is called process based multitasking.

Ex: while typing a java program in the editor we can listen audio songs from same system at the same time we can download a file from Net all these tasks will be executed simultaneously and independent of each other hence it is process based multitasking. Process based multitasking is best suitable at OS Level.

### Thread based multitasking

Executing several tasks simultaneously where each task is a multi-tasking part of the same program is called thread based multitasking and each independent part is called a thread.

→ Thread based multitasking is best suitable at programmatic level



→ Whether it is process based or thread based the main objective of multitasking is to reduce response time of the system and to improve performance.

\* The main important application areas of multithreading are to develop

- ① multimedia graphics
- ② to develop animations
- ③ to develop videogames
- ④ to develop web servers & browsers etc.

When compared with old languages developing multi-threaded applications in java is very easy.

because java provides inbuilt support for multithreading

with rich API (Thread, Runnable, ThreadGroup, ...)

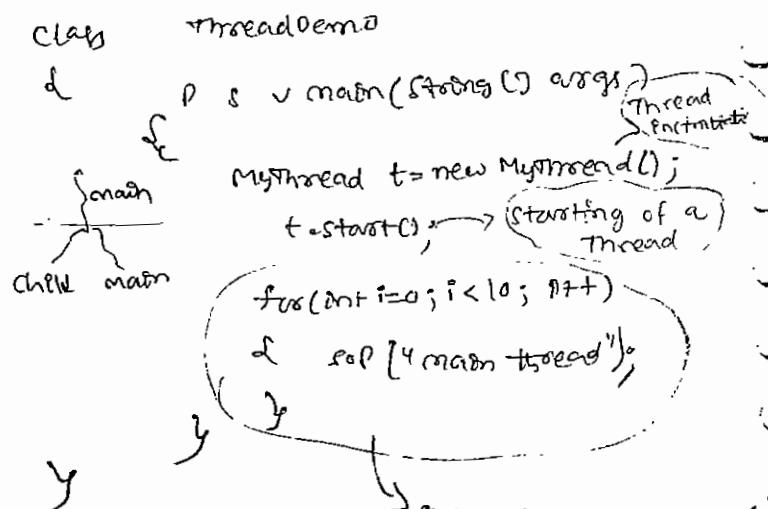
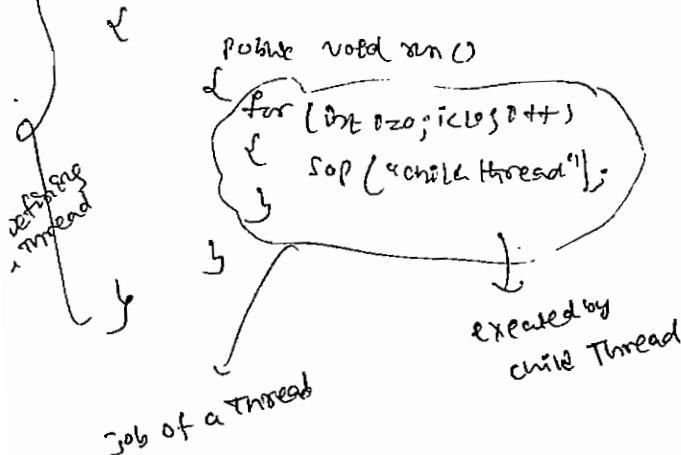
## ② Defining a Thread

We can define a thread in the following ② ways

- ① By extending Thread class
- ② By implementing Runnable interface

### ① By extending Thread class

Class MyThread extends Thread



## Code 16      Thread scheduler

→ It is the part of JVM

→ It is responsible to schedule threads i.e. if multiple threads are waiting to get the chance of execution then in which order threads will be executed is decided by thread scheduler.

→ We can't expect exact algorithm followed by thread scheduler as it is carried from JVM to JVM hence we can't expect threads execution order and exact output hence whenever situation comes to multithreading there is no guarantee for exact output but we can provide several possible outputs.

## Possibility I

main thread  
main thread

Child Thread  
child Thread

6347 2500-0000000000000000

Possibility 2

child Thread  
child Thread

↓  
MainThread  
mainThread

 The following are various possible outputs for the above program

Possibility - 3

```

graph TD
    MT[Main Thread] --- CT[Child Thread]
    subgraph CT
        direction TB
        MT --- MTR[Main Thread]
        MTR --- CTR[Child Thread]
        CTR --- CTR[Child Thread]
    end

```

The diagram illustrates the thread hierarchy. The main thread is at the top. A child thread originates from it. This child thread contains another main thread, which in turn has a child thread of its own.

## Possibility-4

```

graph TD
    ChildThread --- MainThread
    ChildThread --- ChildThread
    MainThread --- MainThread
    MainThread --- ChildThread

```

The diagram illustrates the relationship between threads. It shows two main nodes: "Child Thread" and "Main Thread". Arrows indicate connections between them: one arrow from Child Thread to Main Thread, one arrow from Child Thread to another Child Thread, one arrow from Main Thread to Main Thread, and one arrow from Main Thread to Child Thread.

## Cafe 2.

Difference b/w `t.start()` and `t.run()`

In the case of `t.start()` a new thread will be created which is responsible for the execution of `run()` method but in the case of `t.run()` a new thread won't be created and `run()` method will be executed just like a normal method call by main thread. hence in the above program if we replace `t.start()` with `t.run()` then the output will be

then the output is child thread  
child thread 10 times;

This total output produced by only main thread

Main thread  
Main thread 10 times

### Cafe 3:

Importance of Thread class start() method

Thread class `start()` method is responsible to register the thread with Thread Scheduler and all other mandatory activities hence without executing Thread class `start()` method there is no chance of starting a new thread in Java due to this Thread class `start()` method is considered as heart of multithreading.

### Start()

- 1. Register with this thread with Thread Scheduler;
- 2. Perform all other mandatory activities
- 3. Invoke run() method

}

### Case 4:

Overloading of run() method

Overloading of run() method is always possible but Thread class start() method can invoke no argument run() method.

The other overloaded method we have to call explicitly like a normal method call.

Class MyThread extends Thread

```

    {
        public void run()
        {
            System.out.println("no-arg run");
        }
        public void run(int i)
        {
            System.out.println("int arg run");
        }
    }
  
```

overloaded methods

```

    class ThreadDemo
    {
        public static void main(String[] args)
        {
            MyThread t = new MyThread();
            t.start();
        }
    }
  
```

O/P: No-arg run

### Case 5:

If we are not overriding run() method

If we are not overriding run() method then then thread class run() method will be executed which has empty implementation hence we won't get any output

Ques

Class MyThread extends Thread

```

    {
        class Test
        {
            public static void main(String[] args)
            {
                MyThread t = new MyThread();
                t.start();
            }
        }
    }
  
```

O/P: no output

### Notes

It is highly recommended to override run() method otherwise don't go for multithreading concept.

### Case 6: Overriding of start() method

If we override start() method then our start() method will be executed just like a normal method call and new thread won't be created.

Ex:-

```
class MyThread extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

Output: start method  
run method

Produced by  
only one  
thread

Class Test

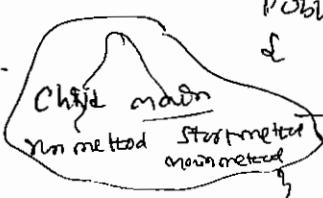
```
{ void main (String [] args)
{
    MyThread t = new MyThread();
    t.start();
    System.out.println("main method");
}}
```

Note: It is not recommended to override start() method otherwise don't go for multithreading concept.

### Case 7:

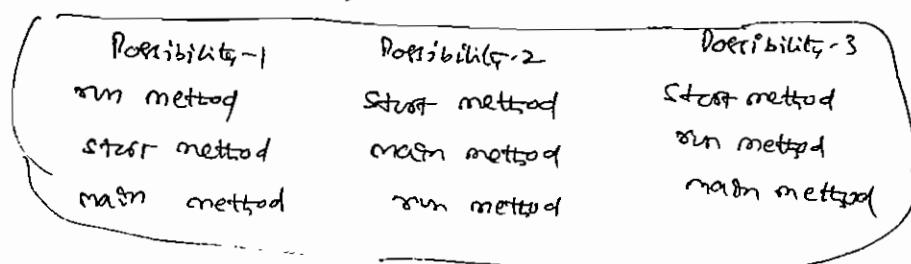
Class MyThread extends Thread

```
{ public void start()
{
    super.start();
    System.out.println("start method");
}
public void run()
{
    System.out.println("run method");
}}
```



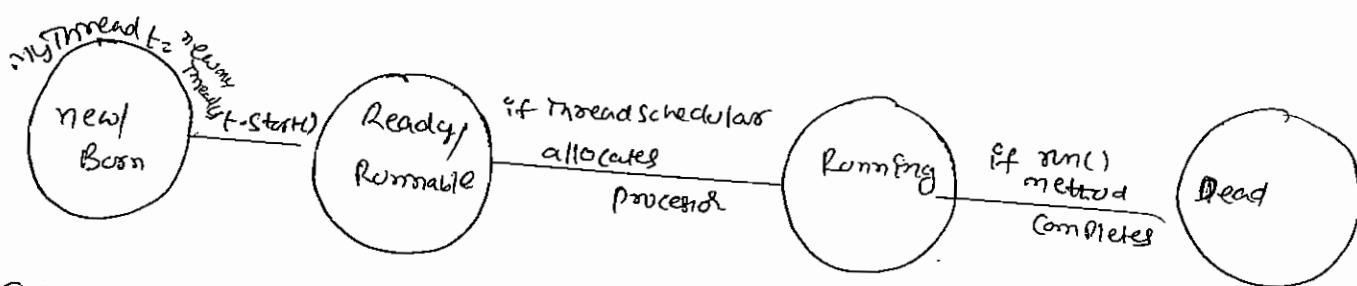
Class Test

```
{ void main (String [] args)
{
    MyThread t = new MyThread();
    t.start();
    System.out.println("main method");
}}
```



## Case 8

### Thread life cycle



## Case 9

After starting a thread if we are trying to restart the same thread then we will get RuntimeException  
IllegalThreadStateException

### Sol

Class Test

```

class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        System.out.println("main method");
        t.start();
    }
}
  
```

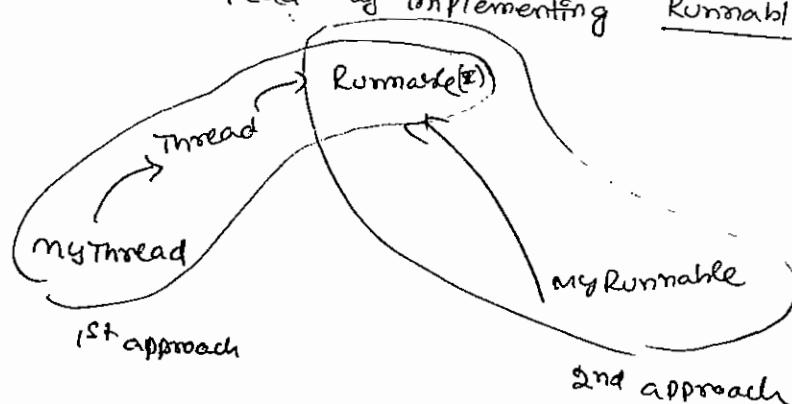
R.E: IllegalThreadStateException

②

Defining a thread by implementing  
Runnable Interface  
 We can define a thread by implementing

Runnable Interface

Runnable Interface



→ Runnable interface present in java.lang package and it contains only one method run() method

Public void run()

```

class MyRunnable implements Runnable
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.println("child thread");
        }
    }
}

```

executed by child Thread

defining thread

Job of a thread

100%

```

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start(); // Target Runnable
        for(int i=0; i<10; i++)
        {
            System.out.println("main thread");
        }
    }
}

```

executed by main thread

- We will get mixed output and we can't tell exact output

### Case Study

```

MyRunnable r = new MyRunnable();
Thread t1 = new Thread();
Thread t2 = new Thread(r);
}

```

#### Case 1:

t1.start();

A new thread will be created and which is responsible for the execution of Thread class run() method, which has empty implementation.

#### Case 2:

t1.run();

No new thread will be created and Thread class run() method will be executed just like a normal method call.

#### Case 3:

t2.start();

A new thread will be created which is responsible for the execution of MyRunnable class run() method.

#### Case 4:

t2.run();

A new thread won't be created and MyRunnable run() method will be executed just like a normal method call.

#### Case 5:

r.start();

We will get compilation error saying MyRunnable class doesn't have start capability.

Case 5: can not find symbol  
symbol: method start()  
location: class MyRunnable

Case 6: ~~start()~~

~~start()~~; no new thread will be created and MyRunnable run() method will be executed like normal method call

3rd class

Which approach is best to define a Thread

Among two ways of defining a thread implements Runnable approach is recommended

- In the first approach our class always extends Thread class, there is no chance of extending any other class hence we are missing inheritance benefit.
- but in the second approach while implementing Runnable interface we can extend any other class hence we won't miss any inheritance benefit because of above reason implementing Runnable interface approach is recommended than extending Thread class.

### Thread Class Constructors

- ① Thread t = new Thread();
- ② Thread t = new Thread(Runnable r);
- ③ Thread t = new Thread(ThreadGroup g, Runnable r, String name);
- ④ Thread t = new Thread(ThreadGroup g, String name);
- ⑤ Thread t = new Thread(Runnable r, String name);
- ⑥ Thread t = new Thread(ThreadGroup g, String name);
- ⑦ Thread t = new Thread(ThreadGroup g, Runnable r);
- ⑧ Thread t = new Thread(ThreadGroup g, Runnable r, String name);

Durga's approach to define a thread (Not recommended to use) stacksize;

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");  
    }  
}  
} off: Child Thread | main thread  
main thread | child thread
```

### (3) Getting & Setting Name of a Thread

Every thread in java has some name it may be default name generated by JVM (or) customized name provided by programmer

We can get & set name of a Thread by using the following 2 methods of Thread class

(1) Public final String getName()

(2) Public final void setName (String name)

Ex6

```
class MyThread extends Thread
{
    class Test
    {
        public static void main (String [] args)
        {
            System.out.println (Thread.currentThread().getName()); // main
            MyThread t = new MyThread ();
            System.out.println (t.getName()); // Thread-0
            Thread.currentThread().setName ("Pawan Kalyan");
            System.out.println (Thread.currentThread().getName()); // Pawan kalyan
        }
    }
}
```

Note: We can get current executing thread object by using Thread.currentThread() method

Ex7

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println ("run method Executed by Thread " + Thread.currentThread().getName());
    }
}
class Test
{
    public static void main (String [] args)
    {
        MyThread t = new MyThread();
        t.start();
        System.out.println ("main method Executed by Thread " + Thread.currentThread());
    }
}
```

O/P7

main method Executed by Thread 5 main  
run method Executed by Thread 6 Thread-0

## ④ Thread Priorities

Every thread in java has some priority it may be default priority generated by JVM or customized priority provided by programmer.

The valid range of thread priorities is 1 - 10 where '1' is min priority and '10' is max priority.

Thread class defines the following constants to represent some standard priorities

Thread.MIN_PRIORITY → 1
Thread.NORM_PRIORITY → 5
Thread.MAX_PRIORITY → 10

Ques

Which of the following are valid Thread priorities in java.

- 0 X
- 1 ✓
- 10 ✓

Thread.LOW\_PRIORITY X

Thread.HIGH\_PRIORITY X

Thread.~~MIN~~\_PRIORITY ✓

Thread.NORM\_PRIORITY ✓

- Thread Scheduler will use priorities while allocating processor.
- The thread which is having highest priority will get chance first
- If two threads having same priority then we can't expect exact execution order it depends on Thread Scheduler

Thread class defines the following methods to set & get priority of a thread

public final int <del>getPriority()</del> <u>getPriority()</u>
--

public final void setPriority (int p)
---------------------------------------

Allowed value range : 1 to 10

Otherwise R.E: IllegalArgumentExcetion

Eg: t.setPriority (7); ✓

t.setPriority (17); X R.E IllegalArgumentExcetion

## default Priority

(26)

The default Priority only for the main thread is 5 but for all remaining threads default priority will be inherited from parent to child i.e. whatever priority parent thread has the same priority will be there for the child thread.

```
Ex6 class MyThread extends Thread
{
    } class Test
    {
        public static void main (String [] args)
        {
            System.out.println(Thread.currentThread().getPriority()); // 5
            // Thread.currentThread().setPriority (15); R.E. IllegalArgumentException
            // Thread.currentThread().getPriority (7); → ①
            MyThread t = new MyThread();
            System.out.println (t.getPriority()); → ②
        }
    }
```

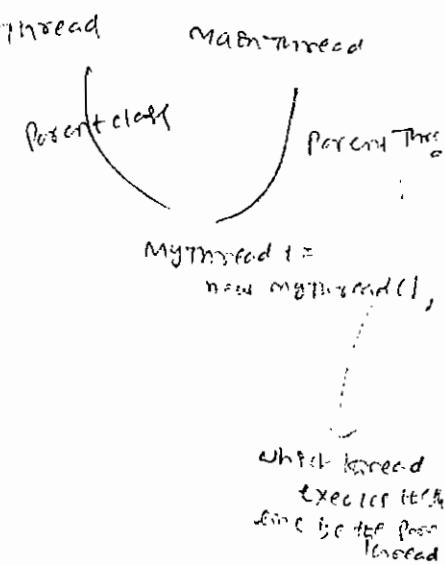
If we comment line ① then child thread priority

will become 5

```
class MyThread extends Thread
{
    public void run ()
    {
        for (int i=0; i<10; i++)
        {
            System.out.println ("child thread");
        }
    }
}
```

```
class ThreadPrioritiesDemo
{
    public static void main (String [] args)
    {
        MyThread t = new MyThread();
        t.setPriority (10); → ①
        t.start ();
        for (int i=0; i<10; i++)
        {
            System.out.println ("main thread");
        }
    }
}
```

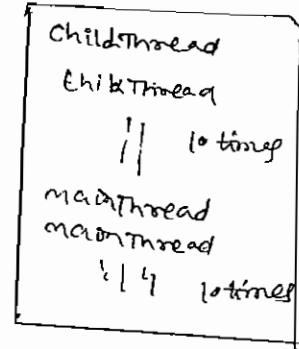
If we are commenting line ① then both main & child threads have the same priority 5 and hence we can't expect execution orders and exact output



If we are not commenting line ① then main thread has the priority 5 and child thread has the priority 6 hence child thread will get the chance first followed by main thread in this case O/P is

### Note

Some platforms won't provide proper support for thread priorities



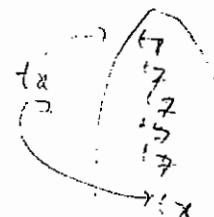
[30-08-14]

⑤ The methods to prevent thread execution  
We can prevent a thread execution by using the following methods

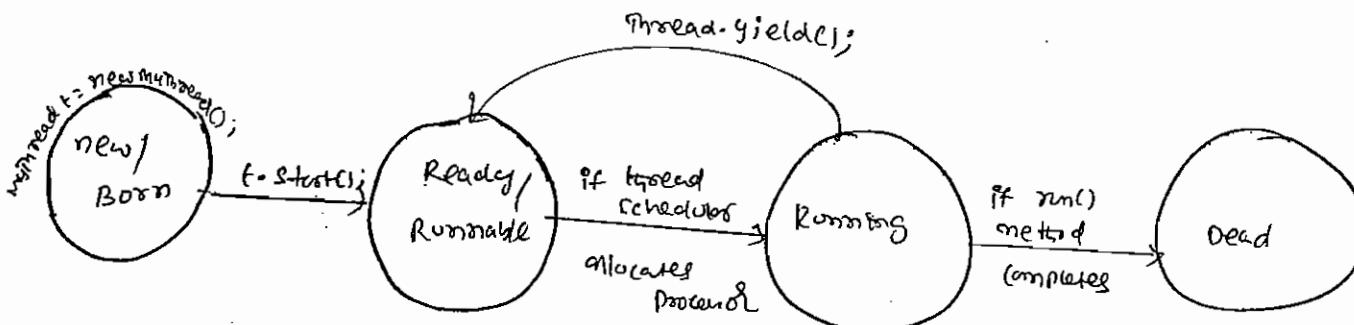
- ① yield()
- ② join()
- ③ sleep()

### ① yield()

- ① yield() method causes to ~~pause~~ current executing thread to give the chance for remaining waiting threads of same priority
- ② If there is no waiting thread (or) all waiting threads have low priority then same thread (or) continue its execution
- ③ If multiple threads are waiting with same priority then which waiting thread will get the chance we can't expect it depends on thread scheduler.
- ④ The thread which is yielded, when it will get the chance once again it depends on thread scheduler and we can't expect exactly.



Public static native void yield();



```

class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.println("child thread");
            Thread.yield();
        }
    }
}

```

class ThreadYieldDemo

```

public class ThreadYieldDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        for(int i=0; i<10; i++)
        {
            System.out.println("main thread");
        }
    }
}

```

- \* In the above Program if we are commenting line ① then both threads will be executed simultaneously and we can't expect which thread will complete first.
- If we are not commenting line ① then child thread always calls yield() method because of that main thread will get chance more number of times and the chance of completing main thread first is high.

Note: Some platforms won't provide proper support for yield method.

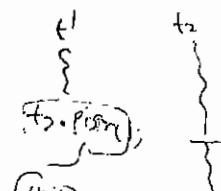
## ② join()

If a thread wants to wait until completing some other thread then we should go for join() method.

For example if a thread  $t_2$  wants to wait until completing  $t_1$  then  $t_1$  has to call  $t_2.join();$

If  $t_1$  executes  $t_2.join()$  then immediately  $t_1$  will be entered waiting state until  $t_2$  completes. Once  $t_2$  completes then  $t_1$  can continue its execution.

join();  
is a blocking  
method  
it is  
advantage  
to  
use



### Ex:

Venu fixing  
Activity  
( $t_1$ )

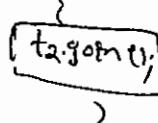
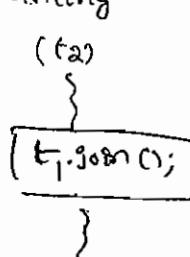
wedding cards  
printing  
( $t_2$ )

wedding cards  
distribution  
( $t_3$ )

Venu fixing  
activity  
( $t_1$ )

wedding  
cards  
printing  
( $t_2$ )

wedding cards  
distribution  
( $t_3$ )

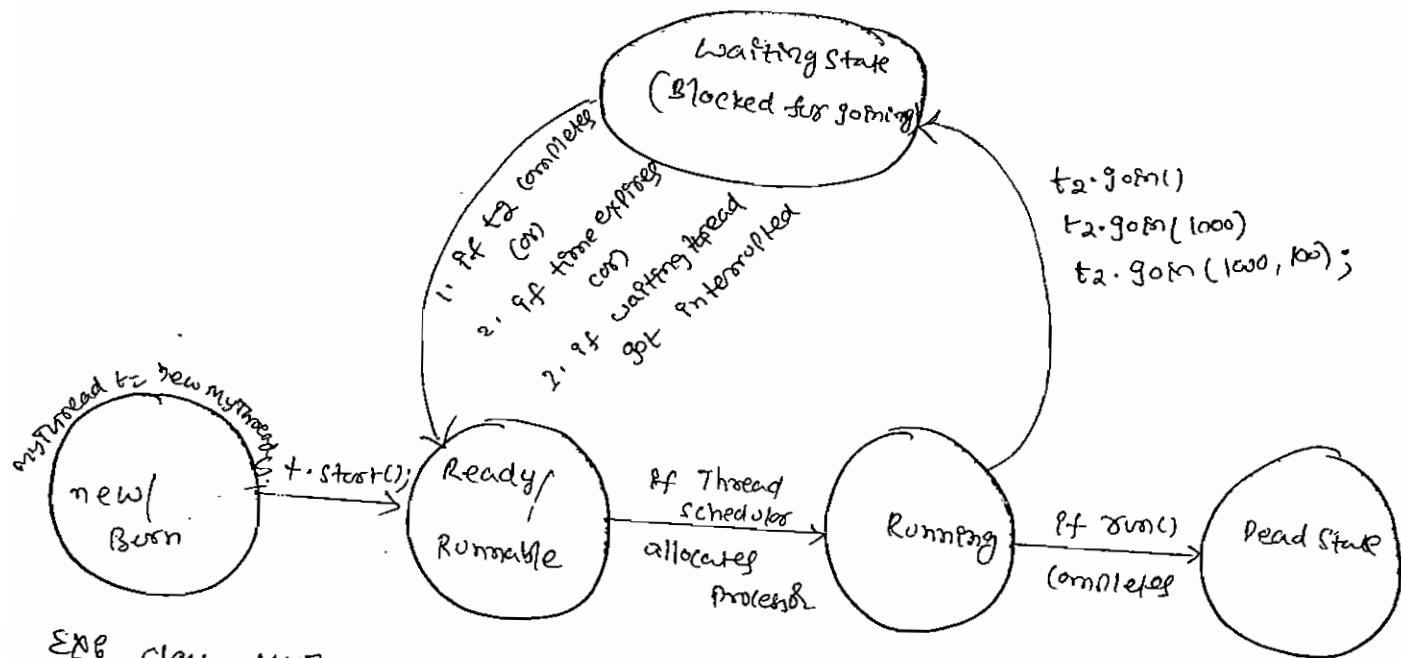


Wedding cards printing thread ( $t_2$ ) has to wait until venu fixing thread ( $t_1$ ) completion. Hence  $t_2$  has to call  $t_1.join()$ .

Wedding cards distribution thread ( $t_3$ ) has to wait until wedding cards printing thread ( $t_2$ ) completion. Hence  $t_3$  has to call  $t_2.join()$ .

- 2.2.e.k
- ① Public final void join() throws InterruptedException
  - ② Public final void join(long ms) throws InterruptedException
  - ③ Public final void join (long ms, int ns) throws InterruptedException

Every join() method throws InterruptedException which is checked exception hence compulsorily we should handle this exception either by using try-catch or by using throws keyword otherwise we will get compilation error.



Ex: class MyThread extends Thread

```

    {
        public void run()
        {
            for(int i=0; i<10; i++)
            {
                System.out.println("Hello Thread");
                try
                {
                    Thread.sleep(200);
                }
                catch(InterruptedException e)
                {
                }
            }
        }
    }
  
```

The code above shows a child thread named `MyThread` that prints "Hello Thread" 10 times and sleeps for 200ms between each print. It catches the `InterruptedException` to prevent the main thread from waiting for it.

### Class ThreadJoinDemo

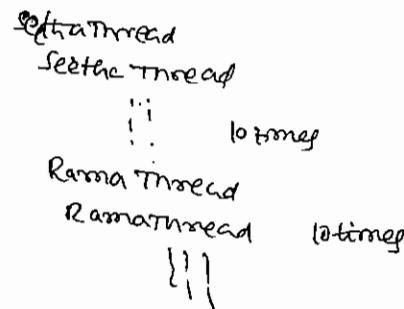
```

public static void main(String[] args) throws InterruptedException
{
    MyThread t = new MyThread();
    t.start();
    t.join(); → ①
    for(int i=0; i<10; i++)
    {
        System.out.println("A Main Thread");
    }
}
  
```

The code above demonstrates the use of `join()`. It creates a child thread `MyThread`, starts it, and then calls `join()` on it. This causes the main thread to wait for the child thread to complete before continuing. The main thread prints "A Main Thread" 10 times while the child thread is running.

If we comment line ① then both main & child threads will be executed simultaneously and we can't expect exact output.

If we are not commenting line ① then main thread calls join() method on child thread object hence main thread will wait until completing child thread in this case output is



## 2nd Case

Waiting of child thread until completing main thread.

Ex:

```

class MyThread extends Thread {
    static Thread mt;
    public void run() {
        try {
            mt.join();
        } catch (InterruptedException e) {
        }
        for (int i = 0; i < 10; i++) {
            System.out.println("child thread");
        }
    }
}

class ThreadJoinDemo1 {
    public static void main(String[] args) throws InterruptedException {
        MyThread mt = Thread.currentThread();
        MyThread mt = new MyThread();
        mt.start();
        for (int i = 0; i < 10; i++) {
            System.out.println("main thread");
            Thread.sleep(2000);
        }
    }
}
  
```

In the above example child thread calls join() method on main thread object hence child thread has to wait until completing main thread

In this case output is

o/p : mainthread  
main Thread  
!!!  
child Thread  
child Thread

Case 3:

If main thread calls join() method on child thread object and child thread calls join() method on main thread object then both threads will wait forever and the program will be stucked.  
(This is something like Deadlock)

Case 4:

If a thread calls join() method ~~on~~ same thread itself then the program will be stucked (this is something like Deadlock)

In this case thread has to wait infinite amount of time.

Eg:

Class Test  
{  
    public void main (String [] args) throws InterruptedException  
    {  
        Thread currentThread = Thread.currentThread();  
        currentThread.join();  
    }  
}

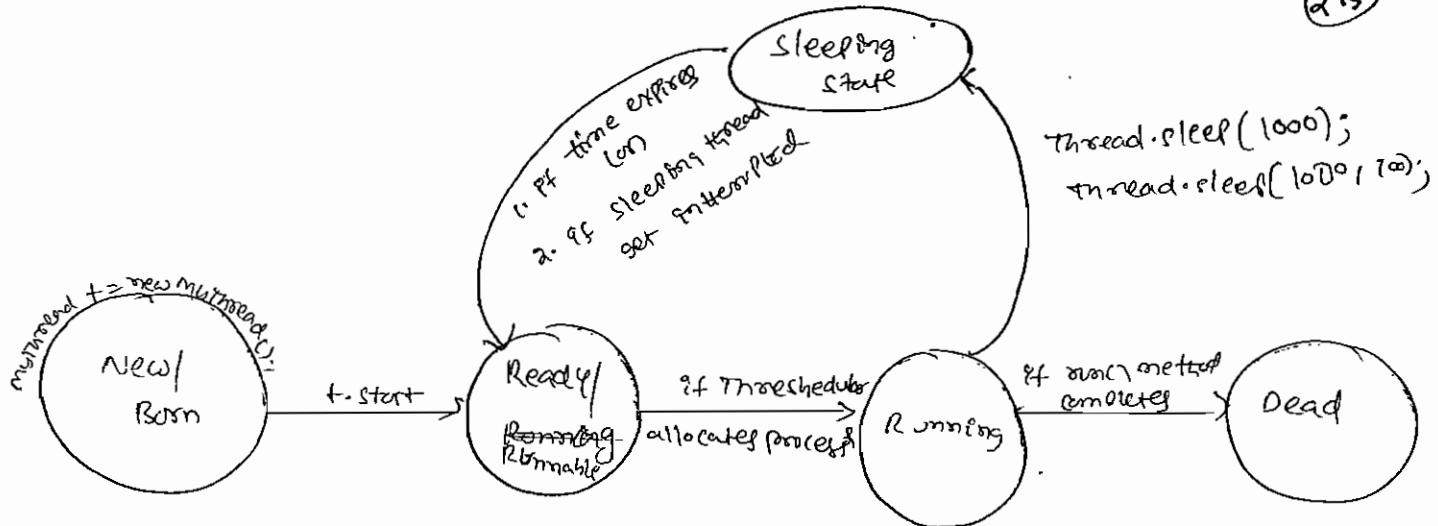
### (3) sleep()

If a thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.

- ① Public static native void sleep (long ms) throws InterruptedException
- ② Public static void sleep (long ms, int ns) throws InterruptedException

Note

Every sleep() method throws InterruptedException, which is checked exception. Hence whenever we are using sleep() method compulsorily we should handle InterruptedException either by try-catch or by throws keyword otherwise we will get compiletime error.



Exe

Class SlideRotator

```

public static void main(String[] args) throws InterruptedException {
    for (int i=1; i<=10; i++) {
        System.out.println("slide -" + i);
        Thread.sleep(5000);
    }
}

```

### Q) How a Thread can Interrupt another Thread

A Thread can Interrupt a sleeping thread (or) waiting thread by using `interrupt()` method of Thread class

`public void interrupt()`

Exe Class MyThread extends Thread

```

public void run() {
    try {
        for (int i=0; i<10; i++) {
            System.out.println("I am Lazy Thread");
            Thread.sleep(2000);
        }
    } catch (InterruptedException e) {
        System.out.println("I got Interrupted");
    }
}

```

## Class ThreadInterruptDemo

```
{  
    public static void main(String[] args)  
    {  
        myThread t = new MyThread();  
        t.start();  
        t.interrupt();  
    }  
    System.out.println("End of main");  
}
```

If we comment line ① then main thread won't interrupt child thread  
In this case child thread will execute for loop 10 times

If we are not commenting line ① then main thread interrupts child thread  
In this case output is  
    0 0 0 End of main thread  
    I am lazy thread  
    I got Interrupted

Ex

### Notes

- ① Whenever we are calling interrupt() method if the target thread not in sleeping state or waiting state then there is no impact of interrupt call.  
    If the target thread entered into sleeping or waiting state then immediately interrupt call will be waited until target thread entered into sleeping or waiting state.
- ② If the target thread entered into sleeping or waiting state then immediately interrupt call will interrupt the target thread.
- ③ If the target thread never entered into sleeping (or) waiting state for its life time then there is no impact of interrupt call.  
    This is the only case where interrupt call will be wasted.

### Ex6

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        for(int i=0; i<=10000; i++)  
        {  
            System.out.println("I am Lazy thread... " + i);  
        }  
        System.out.println("I am entering into sleeping state");  
        try{  
            Thread.sleep(10000);  
        } catch(InterruptedException e)  
        {  
            System.out.println("I got Interrupted");  
        }  
    }  
}
```

## (2)

### Class ThreadSleepDemo

```

    {
        public static void main (String [] args)
        {
            MyThread t = new MyThread ();
            t.start ();
            t.interrupt ();
        }
    }
}

```

System.out.println ("End of main thread");

\* In the above example interrupt call waited until child thread completes forloop 10000 times (ten thousand times)

Comparison table of yield(), join() & sleep() methods

Property	yield()	join()	sleep()
(1) Purpose	If a thread wants to pause its execution to give the chance for remaining threads of same priority then we should go for yield() method.	If a thread wants to wait until completing some other thread then we should go for join() method	If a thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
(2) Is it overloaded	No	Yes	Yes
(3) Is it final	No	Yes	No
(4) Is it throws InterruptedException	No	Yes	Yes
(5) Is it native method	Yes	No	sleep (long ms) → native sleep (long ms, <del>int</del> ms) ↓ non-native
(6) Is it static	Yes	No	Yes

⑥

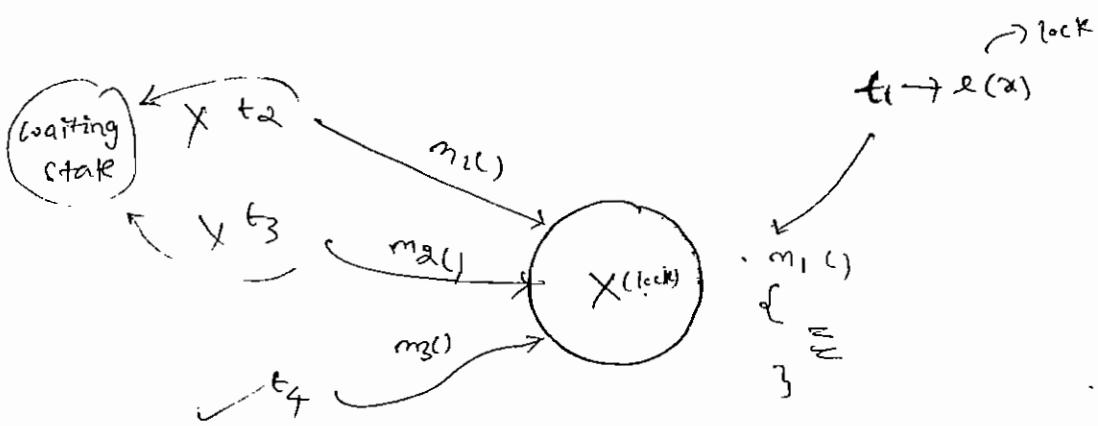
## Synchronization

- Synchronized is the modifier applicable only for methods & blocks but not for classes and variables.
  - if multiple threads are trying to operate simultaneously on the same Java Object then there may be a chance of data inconsistency problem.
  - to overcome this problem we should go for synchronized keyword.
  - If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given Object so that data inconsistency problem will be resolved.
  - The main advantage of synchronized keyword is we can resolve data inconsistency problems but the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems hence if there is no specific requirement then it is not recommended to use synchronized keyword.
- ↳ Internally synchronization concept is implemented by using lock.
- every object in java has a unique lock.
  - whenever we are using synchronized keyword then only lock concept will come into the picture.
- If a thread wants to execute synchronized method on the given object first it has to get lock of that object. once thread got the lock then it is allowed to execute any synchronized method on that object. once method execution completes automatically thread releases lock.
- acquiring & releasing lock internally takes care by JVM and programmer not responsible for this activity.

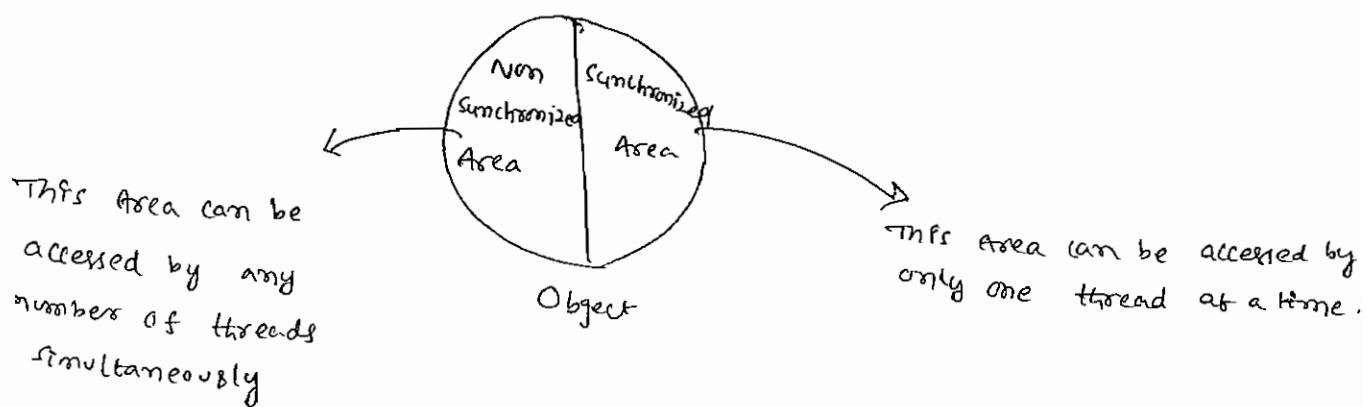
while a thread executing synchronized method on the given object the remaining threads are not allowed to execute any synchronized method simultaneously on the same object but remaining threads are allowed to execute non synchronized methods simultaneously.

Eg:

```
class X
{
    synchronized
    {
        m1()
        synchronized m2()
        m3()
    }
}
```



→ Lock concept is implemented based on object but not based on method



Class X

Synchronized Area

Whenever we are  
Performing update operation  
(add | remove | delete | replace)

i.e. where state of object changing

non-synchronized Area

where ever object state  
won't be changed like  
read operation

↳

Ex6

## Class ReservationSystem

{

Non-synchronized checkAvailability()

{

    ||| just read operator

}

Synchronized bookTicket()

{

    ||| update

}

Ex28

## Class Display

{

    public synchronized void wish (String name)

    { for (int i=0; i<10; i++)

        {

            System.out.println ("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;

    }

    public void run()

{

        d.wish (name);

}

Class SynchronizedDemo

{

    public static void main (String [] args)

{

        Display d = new Display ();

        MyThread t1 = new MyThread (d, "Phong");

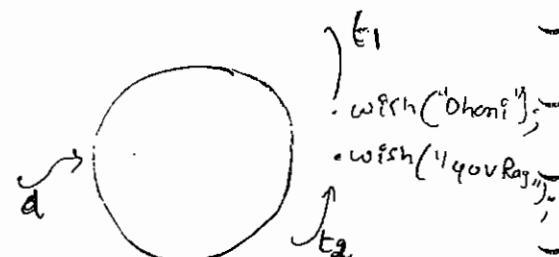
        MyThread t2 = new MyThread (d, "YuvRaj");

        t1.start();

        t2.start();

}

}



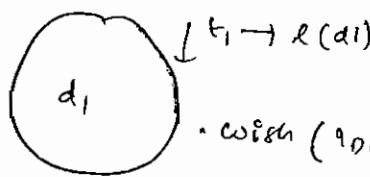
- are not synchronized then at a time
- if we declare wish() method as synchronized then at a time both threads will be executed simultaneously and hence we will get irregular output.
- O/P: GoodMorning: GoodMorning : YUVRAJ  
GoodMorning : Dhoni  
GoodMorning : YUVRAJ  
!!
- If we declare wish() method as synchronized then at a time only one thread is allowed to execute wish() method and given display object hence we will get regular output.

O/P: GoodMorning : Dhoni  
Good Morning : Dhoni  
!! 10 times

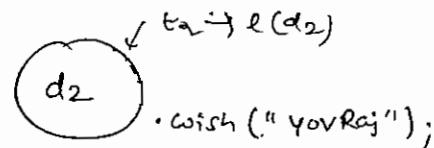
Good Morning : YUVRAJ  
Good Morning : YUVRAJ      vice versa  
!! 10 times

ClassCase Study

```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "YUVRAJ");
t1.start();
t2.start();
```



wish ("Dhoni");



wish ("YUVRAJ");

Even though wish method is synchronized we will get irregular output because threads are operating on different Java objects.

Reason / Conclusion

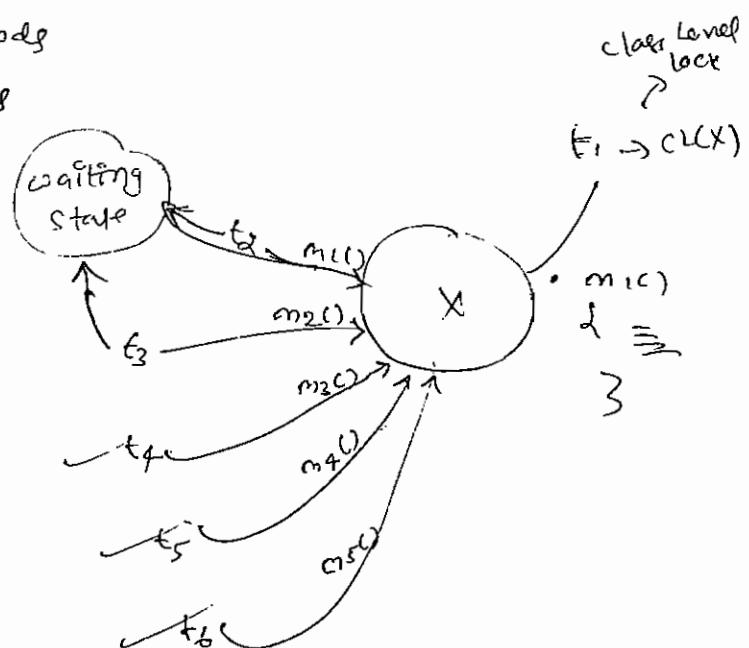
- If multiple threads are operating on same Java object then synchronization is required.
- If multiple threads are operating on multiple Java objects then synchronization is not required.

## → Class Level Lock

Every class object has a unique lock which is also nothing but class level lock.

- if a thread wants to execute a static synchronized method then thread required class level lock.
- Once thread got classlevel lock then it is allowed to execute any static synchronized method of that class.
- Once method execution completes automatically thread releases the lock.
- while a thread executing static synchronized() method the remaining threads are not allowed to execute any static synchronized () method of that class simultaneously. but remaining threads are allowed to execute the following methods simultaneously
  - ① normal static methods
  - ② synchronized instance methods
  - ③ normal instance methods

Ex 6  
Class X  
↳  
    Static Synchronized m<sub>1</sub>( )  
    Static Synchronized m<sub>2</sub>( )  
    Static m<sub>3</sub>( )  
    Synchronized m<sub>4</sub>( )  
    ↳  
        m<sub>5</sub>( )



Ex 3:

Class Display  
↳  
    Public synchronized void display( )  
    {  
        for (int i=1; i<=10; i++)  
        {  
            System.out.println(i);  
            try  
            {  
                Thread.sleep(2000);  
            }  
            catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }

```

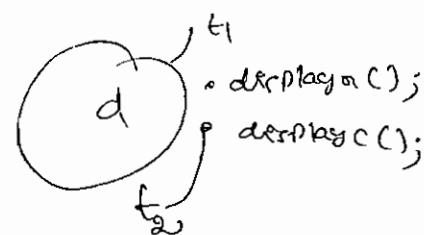
public synchronized void display()
{
    for (int i=65; i<=75; i++)
    {
        sop ((char)i);
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e) {}
    }
}

class MyThread1 extends Thread
{
    Display d;
    MyThread1 (Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.display();
    }
}

class MyThread2 extends Thread
{
    Display d;
    MyThread2 (Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.display();
    }
}

class SynchronizedDemo
{
    public static void main (String [] args)
    {
        Display d = new Display();
        MyThread t1 = new MyThread1(d);
        MyThread t2 = new MyThread2(d);
        t1.start();
        t2.start();
    }
}

```



31-08-16

## Synchronized Block

- If very few lines of the code required synchronization then it is not recommended to declare entire method as synchronized we have to enclose those few lines of the code by using synchronized block
- The main advantage of synchronized block over synchronized method is it reduces waiting time of threads and improve performance of the system.

we can declare synchronized block as follows

- ① To get lock on current object
- ② To get lock on particular object 'b'
- ③ To get class level lock;

① To get lock on current object

Synchronized (this)

{

=

↳ If a thread got lock on current object then only it is allowed to execute this Area.

② To get lock on particular object 'b'

Synchronized (b)

{

=

↳ If a thread got lock on particular object 'b' then only it is allowed to execute this Area.

③ To get Class Level lock

Synchronized (Display.class)

{

=

↳ If a thread got class level lock on "Display" class then only it is allowed to execute this Area.

Ex: class Display

↓

    Public void wish (String name)

{

        ;;;;;; // blank lines or code

    Synchronized (this)

{

        for (int i=0; i<10; i++)

            Log.i ("Good Morning");

```

try
{
    thread.sleep(2000);
}
catch (InterruptedException e)
{
    System.out.println(name);
}

//; // each or 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);
    }
}

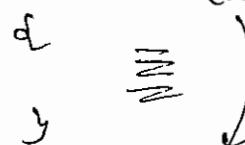
class SynchronizedDemo
{
    public static void main(String[] args)
    {
        Display d = new Display();
        MyThread t1 = new MyThread(d, "Dhoni");
        MyThread t2 = new MyThread(d, "Yuvraj");
        t1.start();
        t2.start();
    }
}

```

→ lock concept applicable for Object types & class types but not primitives hence we can't pass primitive type as argument to synchronized block. otherwise we will get compilation error saying unexpected type found int required reference

Ex: int x=10;

synchronized (x)



CE: unexpected type

found: int

required: reference

## FAQ's

- ① what is synchronized keyword where we can apply?
- ② Explain Advantage of Synchronized keyword?
- ③ Explain disadvantage of Synchronized keyword?
- ④ What is Race condition?  
If multiple threads are operating simultaneously on same java object then there may be a chance of data inconsistency problem. This is called Race Condition.  
We can overcome this problem by using synchronized keyword.
- ⑤ What is Object lock & when it is required?
- ⑥ What is class level lock & when it is required?
- ⑦ What is the difference b/w class level lock & object level lock?  
While a thread executing synchronized method on the given object is the remaining threads are allowed to execute any other synchronized method simultaneously on the same object?  
No
- ⑧ What is synchronized block?
- ⑨ How to declare synchronized block to get lock on current object?
- ⑩ How to declare synchronized block to get Class Level Lock?
- ⑪ What is the advantage of synchronized block over synchronized method?  
Ans) Yes, of course from different objects

Ex: Class X

```
public synchronized void m() {  
    //  
    → here thread has lock on 'X' object
```

```
Y Y = new Y();  
synchronized(Y)  
{  
    //  
    → here thread has locks of X & Y
```

```
Z Z = new Z();
```

```
synchronized(Z)  
{  
    //  
    → here thread has locks of X, Y, Z
```

y i y

y

z

① What is Synchronized Statement? (Interview people created terminology)

The statements present in synchronized methods and synchronized block are called synchronized statements.

## ② Daemon Threads

The threads which are executing in the background are called Daemon threads.

- Eg: 1) Garbage Collector  
2) Signal Dispatcher  
3) Attach Listener .... etc.

→ The main objective of Daemon threads is to provide support for non Daemon thread (main thread).

→ For example if main thread runs with low memory then JVM runs garbage collector to destroy useless objects so that number of bytes of free memory will be improved with this free memory main thread can continue it's execution.

→ Usually ~~the~~ Daemon threads having low priority but based on our requirement Daemon threads can run with high priority also.

→ We can check Daemon nature of a thread by using isDaemon() method of Thread class.

→ We can change Daemon nature of a thread by using setDaemon(boolean b) method.

but changing Daemon nature is possible before starting of a thread only after starting a thread if we are trying to change Daemon nature then we will get RuntimeException saying IllegalThreadStateException.

Default nature of Thread  
By default main thread is always non Daemon and for all remaining threads Daemon nature will be inherited from parent to child. i.e. if the parent thread is Daemon then automatically child thread is also daemon.  
and if the parent thread is non ~~daem~~ daemon then automatically child thread is also non daemon.

Notes It is impossible to change Daemon nature of main thread.  
because which is already started by JVM at beginning

Ex:

```
class MyThread extends Thread  
{  
    class Test  
    {  
        public void main(String[] args)  
        {  
            System.out.println("Thread.currentThread().isDaemon()"); false  
            Thread.currentThread().setDaemon(true); // Re : IllegalThreadStateException  
            MyThread t = new MyThread();  
            System.out.println("t.isDaemon()"); false  
            t.setDaemon(true);  
            System.out.println("t.isDaemon()"); true  
        }  
    }  
}
```

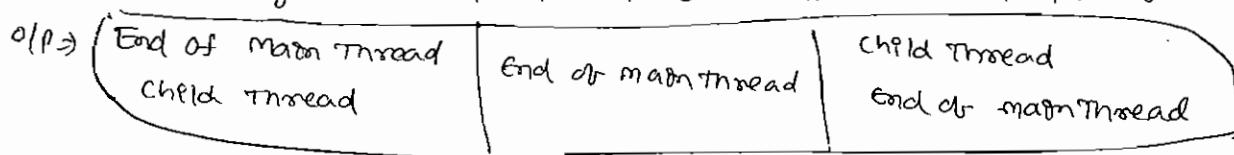
→ whenever last non Daemon terminates automatically all Daemon threads will be terminated irrespective of their position.

Ex:

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        for (int i=0; i<10; i++)  
        {  
            System.out.println("child thread");  
            try  
            {  
                Thread.sleep(2000);  
            } catch (InterruptedException e) {}  
        }  
    }  
}  
  
class DaemonThreadDemo  
{  
    public void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.setDaemon(true); → ①  
        t.start();  
        System.out.println("End of main Thread");  
    }  
}
```

If we are commenting line ① both main & child threads are non-Daemon and hence both threads will be executed until their completion. (289)

If we are ~~not~~ commenting line ① then main thread is non Daemon & child thread is Daemon hence whenever main thread terminates automatically child thread will be terminated in this case output is:



Ques

## 7 Interthread communication

① Two threads can communicate with each other by using `wait()`, `notify()` and `notifyAll()` methods.

The thread which is expecting updation, it has to call `wait()` method then immediately it will enter into waiting state.

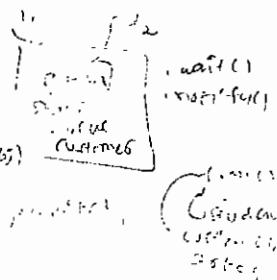
The thread which is performing updation, after performing updation it is responsible to call `notify()` method then waiting thread will get that notification & continue its execution with those updated items.

→ ② `wait()`, `notify()`, `notifyAll()` methods present in Object class but not in Thread class because Thread can call these methods on any java object.

→ ③ To call `wait()`, `notify()`, `notifyAll()` methods on any object, Thread should be owner of that object i.e. the thread should have lock on that object i.e. `Object.wait()`

→ Hence we can call `wait()`, `notify()` & `notifyAll()` methods only from synchronized area.

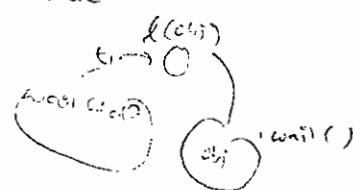
Runtime exception saying IllegalMonitorStateException



↳ Condition  
↳ Waiting  
↳ Notify

→ If a thread calls `wait()` method on any object it immediately releases the lock of that particular object & entered into waiting state.

→ If a thread calls `notify()` method on any object it releases the lock of that object but may not immediately.



⑤ Except `wake()`, `notify()` `notifyAll()` there is no other method where thread releases the lock.

Method	Is Thread releases lock?
<code>yield()</code>	→ No
<code>join()</code>	→ No
<code>sleep()</code>	→ No
<code>wait()</code>	→ Yes
<code>notify()</code>	→ Yes
<code>notifyAll()</code>	→ Yes

SCJP 5.0

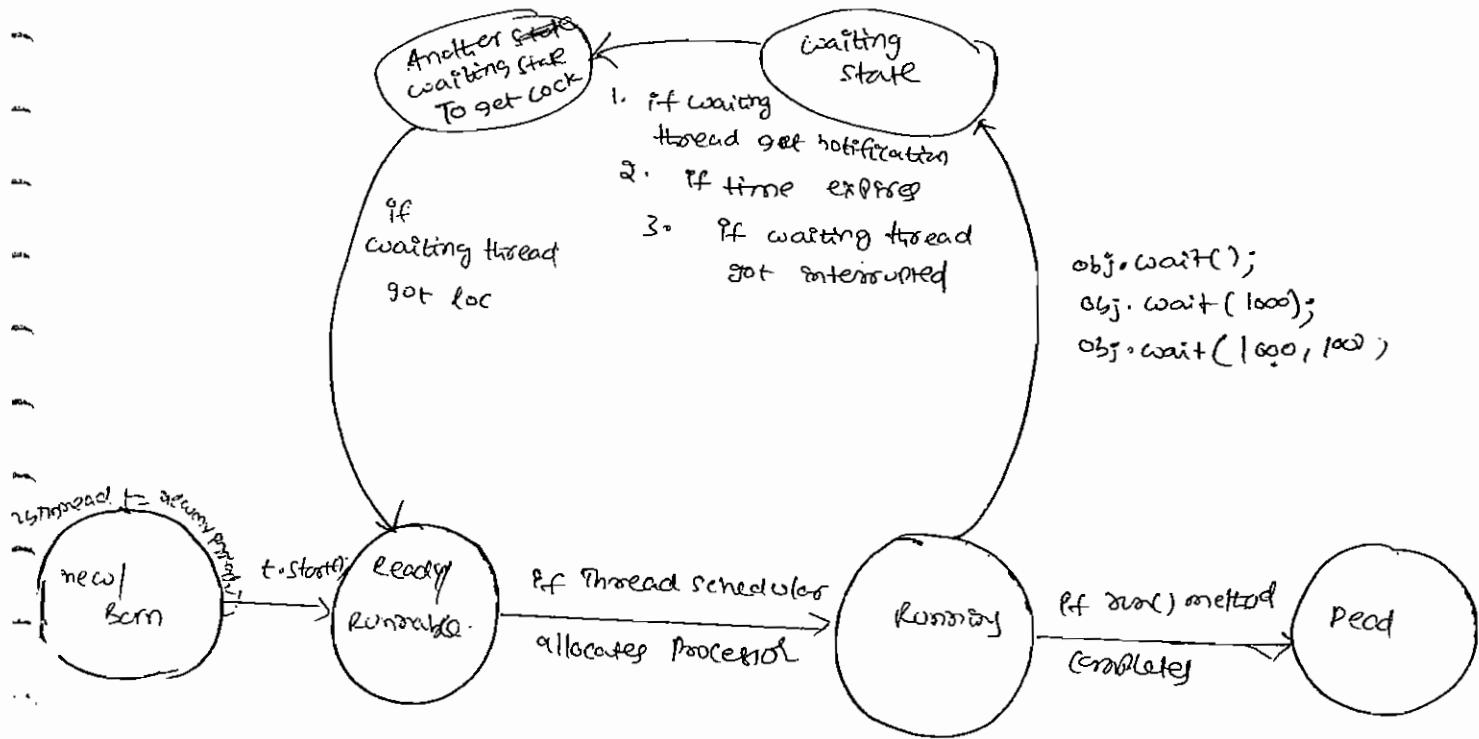
① Which of the following is valid?

- ① If a thread calls `wait()` method immediately it will entered into waiting state without releasing any lock X (invalid)
- ② If a thread calls `wait()` method it releases the lock of that object but may not immediately X (invalid)
- ③ If a thread calls `wait()` method on any object it releases all locks acquired by the thread and immediately entered into waiting state X (invalid)
- ④ If a thread calls `wait()` method on any object it immediately releases the lock of that particular object and entered into waiting state X (invalid)
- ⑤ If a thread calls `notify()` method on any object it immediately releases the lock of the particular object ✓ (valid)
- ⑥ If a thread calls `notify()` method on any object it releases the lock of that object but may not immediately ✓

### Methods

- 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 native void `notifyAll()`

Notes Every `wait()` method throws `InterruptedException` which is checked exception hence whenever we are using `wait()` method compulsorily we should handle this interrupted exception either by try-catch or by throws keyword. otherwise we will get compiletime error.



```

class ThreadA {
  ① s = main(strong[] args) throws exception
  {
    ThreadB b = new ThreadB();
    b.start();
    synchronized(b) { // to get lock on b object
      ② Sop("main thread calling wait method");
      b.wait();
    }
    ④ Sop("main thread got notification");
    ⑤ Sop(b.total);
  }
}

class ThreadB extends Thread {
  int total=0;
  public void run()
  {
    synchronized(this)
    {
      ③ Sop("child thread starts calculation");
    }
  }
}
  
```

```

for(int i=1; i<=100; i++)
{
    total = total + i;
}
(3) → stop("child Thread giving notification");
      this.notify();
}
}

```

off

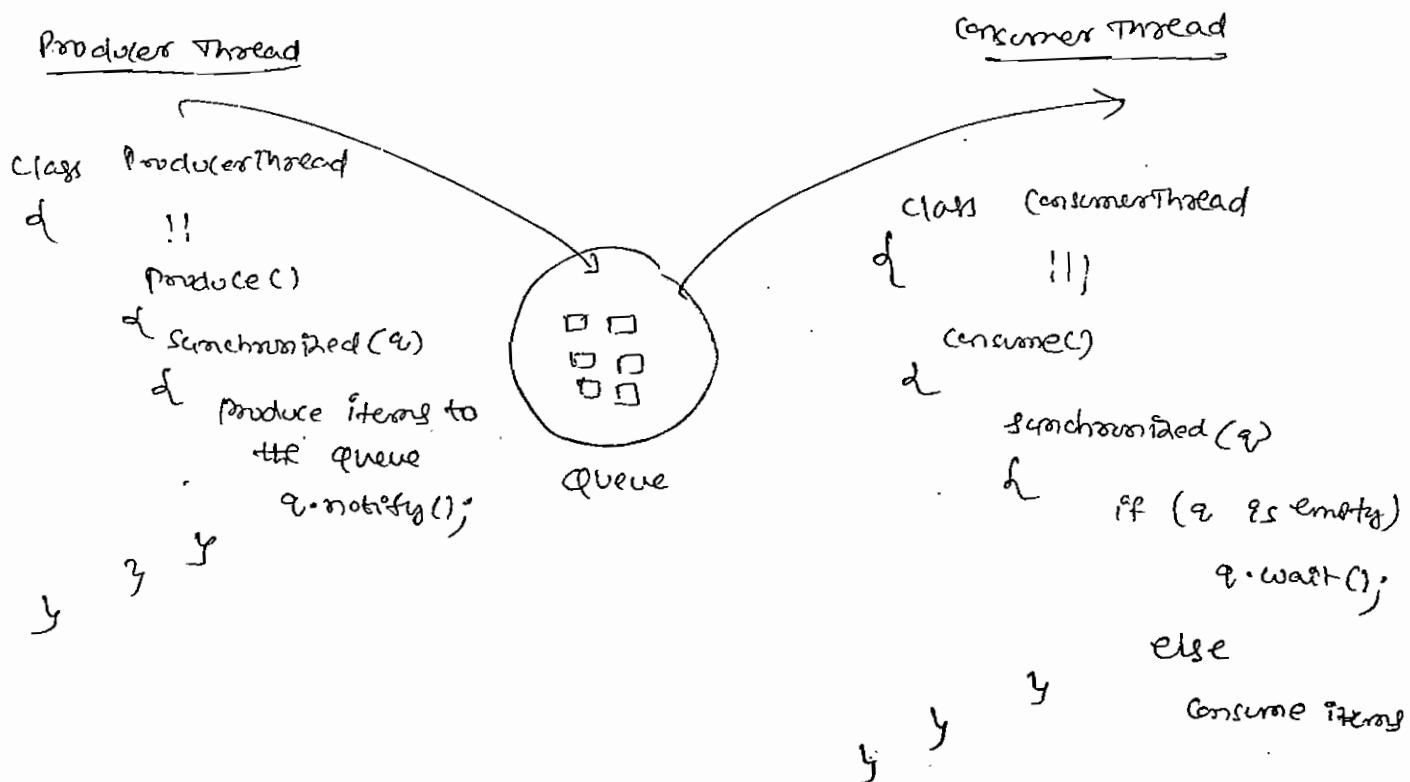
main thread calling wait method  
 child thread starts calculation  
 child thread giving notification  
 main thread got notification  
 so so

## Producers-Consumers Problem

Producer thread is responsible to produce items to the Queue & Consumer thread is responsible to consume items from the queue.

If queue is empty then consumer thread will call wait() method and entered into waiting state.

after producing items to the queue producer thread is responsible to call notify method then waiting consumer will get that notification and continue its execution with updated items.



## Difference b/w notify() and notifyAll()

- ① we can use notify() method to give the notification for only one waiting thread. If multiple threads are waiting then only one thread will be notified and the remaining threads have to wait for further notifications.

- which thread will be notify we can't expect it depends on JVM. (28)
- => we can use `notifyAll()` to give the notification for all waiting threads to a particular object then only one thread even though multiple threads notify but execution will be performed one by one because threads required lock and only one lock is available.

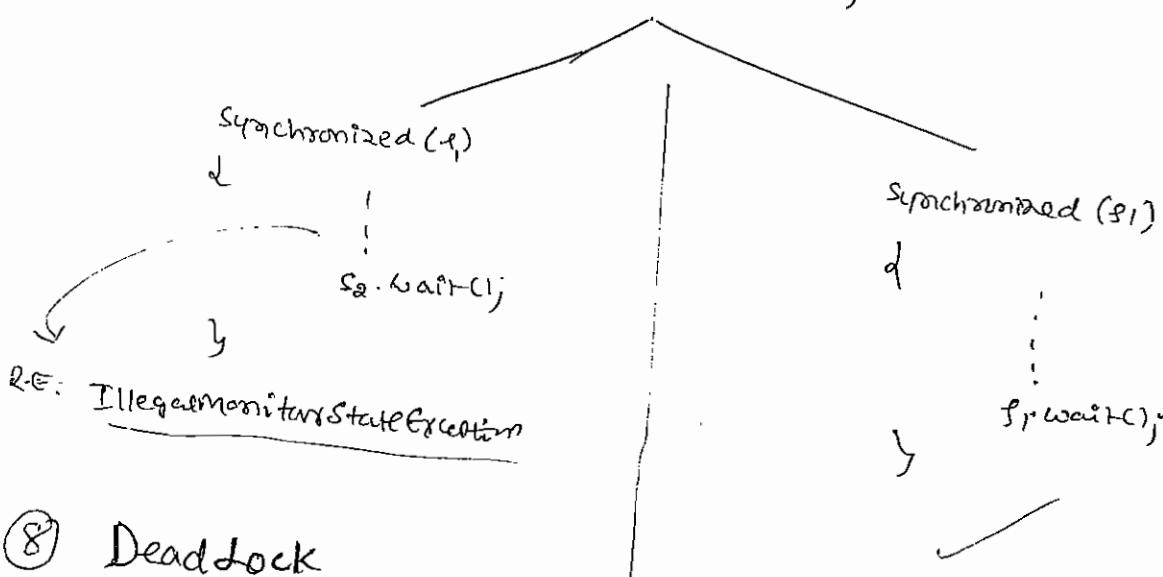
### SCP Notes

on which object we are calling `wait()` method thread required ~~the~~ lock of <sup>that</sup> particular object for example if we are calling `wait()` method on `s1` then we have to ~~get~~ ~~wait() method on s<sub>1</sub> but~~ ~~not on s<sub>2</sub>~~ lock of `s1` object but not `s2` object

### Ex:

`Stack s1 = new Stack();`

`Stack s2 = new Stack();`



## → ⑧ DeadLock

If two threads are waiting for each other forever such type of infinite waiting is called DeadLock.

- `Synchronized` keyword is the only reason for deadlock situation. hence while using `synchronized` keyword we have to take special care
- there are no resolution technique for deadlock but several prevention techniques are available

### Ex:

```

class A
{
    public synchronized void d1(B b)
    {
        System.out.println("Thread 1 starts execution over d1() method");
        Thread.sleep(6000);
    }
}
  
```

catch (InterruptedException e) { }

    System.out.println("Thread1 trying to call B's last()");

    b.last();

    public synchronized void last()

        System.out.println("Inside A, this is last() method");

Class B

{ public synchronized void d2(A a)

    System.out.println("Thread2 starts execution of d2() method");

    try { Thread.sleep(6000); }

    y

    catch (InterruptedException e) { }

        System.out.println("Thread2 trying to call A's last()");

        a.last();

    public synchronized void last()

        System.out.println("Inside B, this is last() method");

y

Class DeadLock1 extends Thread

{

    A a = new A();

    B b = new B();

    public void m1()

{

        t1.start();

        a.d1(b); // This line executed by main thread

,

    public void m2()

{

        b.d2(a);

        // This line executed by child thread

}

    public static void main(String[] args)

{

        DeadLock1 d = new DeadLock1(); ops

y

y

    d.m1();

    Thread1 starts execution of d1() method

    Thread2 starts execution of d2() method

    Thread2 trying to call A's last()

    Thread 1 trying to call B's last()

In the above program if we remove atleast one synchronized keyword then the program won't enter into deadlock. hence synchronized keyword is the only reason for deadlock situation. due to this while using synchronized keyword we have to take special care.

### DeadLock vs Starvation

- Long waiting of a Thread where waiting never ends is called **Deadlock**
- whereas long waiting of a thread where waiting ends at certain point is called **Starvation**  
for ex: low priority thread has to wait until completing all high priority threads, <sup>it may be long</sup> this waiting but ends at certain point, is called which is nothing but starvation

\* → Java Multithreading concept is implemented by using the following 2 models  
① Green Thread model  
② Native OS model

#### ① Green Thread Model

The thread which is managed completely by JVM without taking underlying OS support is called **Green Thread**.

Very few operating systems like SUN Solaris provides support for Green Thread model. any way Green Thread model is depreciated and not recommended to use.

#### ② Native OS Model

The thread which is managed by the JVM with the help of underlying OS, is called **Native OS Model**

All windows based operating systems provide support for native OS model

### How to stop a thread

We can stop a thread execution by using **stop()** method of Thread class

`Public void stop()`

If we call **stop()** method then immediately the thread entered into Dead State  
any way **stop()** method is depreciated and not recommended to use.

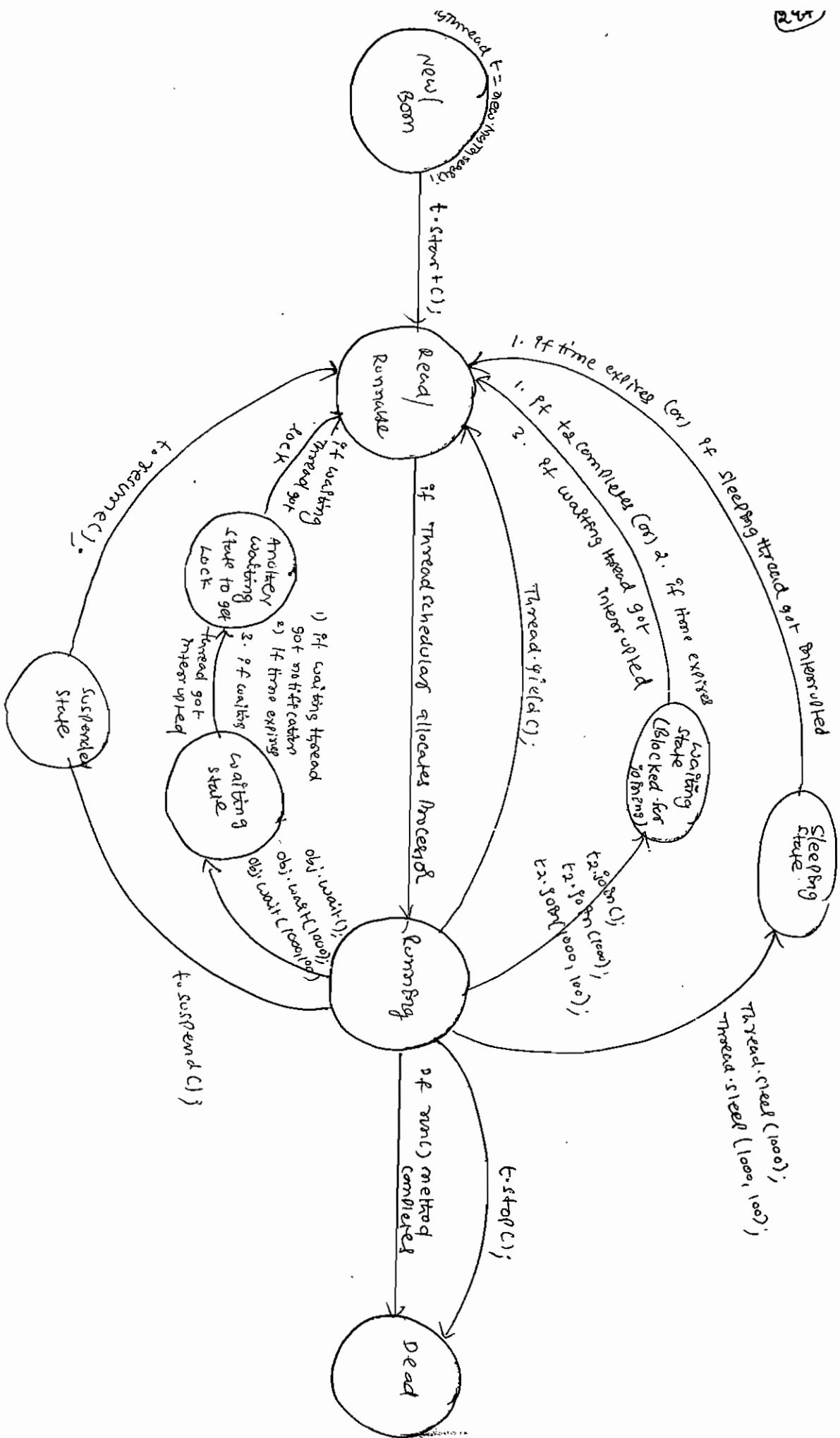
## How to Suspend & Resume of a Thread

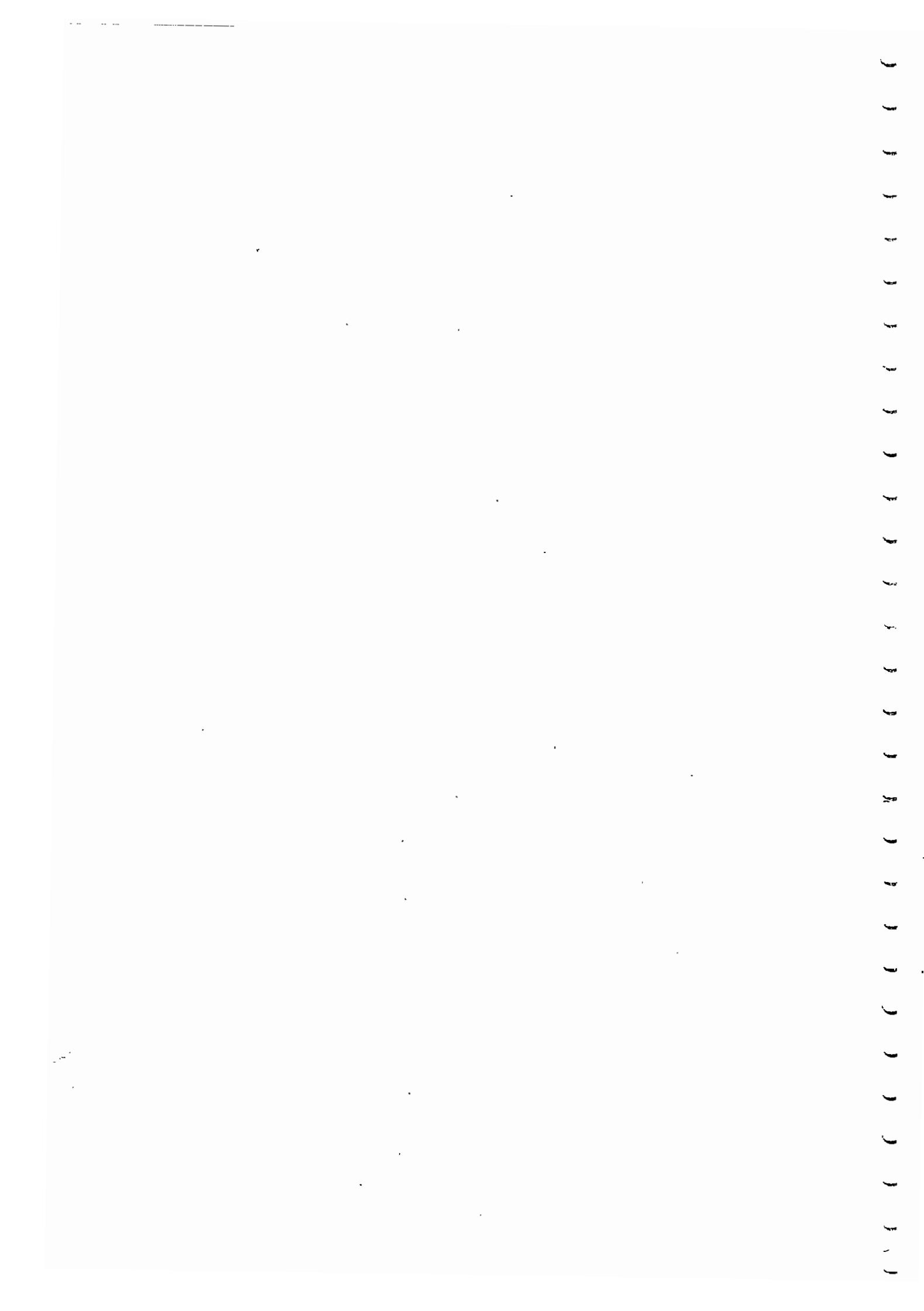
- We can suspend a thread by using `suspend()` method of `Thread class` then immediately the thread will be entered into suspended state.
- We can resume a suspended thread by using `resume()` method of `Thread class` then suspended thread can continue its execution

Public void suspend()

Public void resume()

- Any way these methods are deprecated and not recommended to use.





## Thread Group

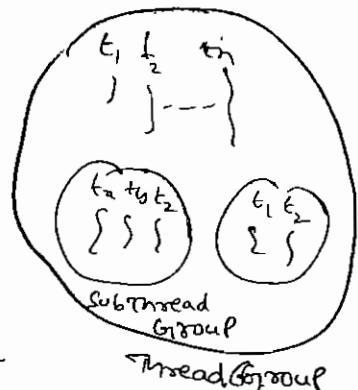
## Multithreading Enhancements

K-12

Based on functionality we can group threads into a single unit which is nothing but Thread Group i.e. ThreadGroup containing a group of threads.

In addition to threads ThreadGroup can also contain sub-thread groups.

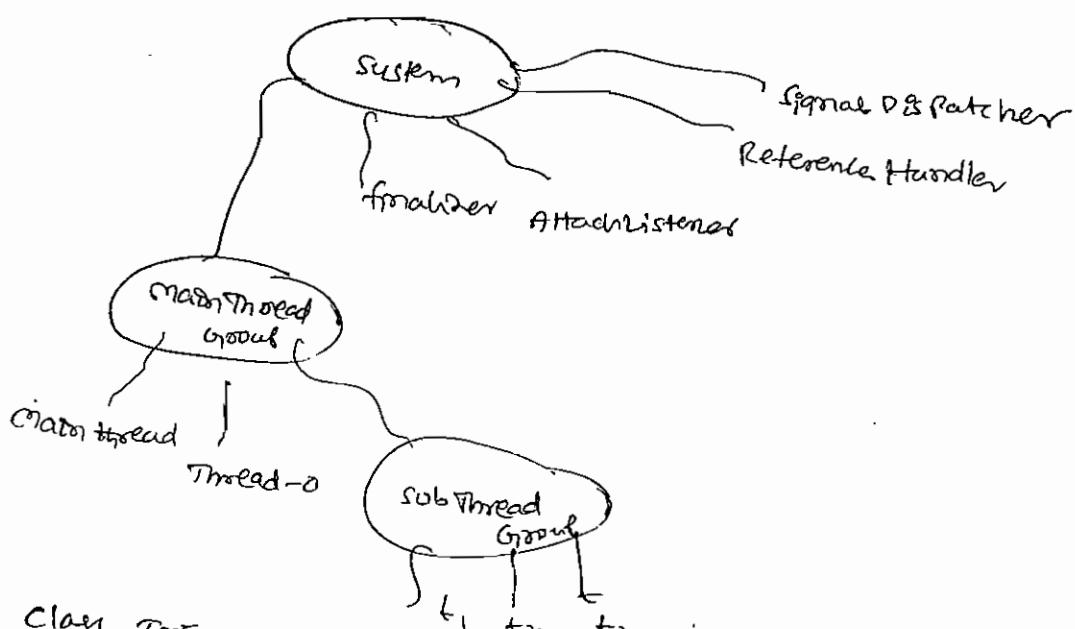
The main advantage of maintaining threads in the form of ThreadGroup is we can perform common operations very easily.



- Every thread in Java belongs to some group
- Main Thread belongs to MainGroup
- Every ThreadGroup in Java is the child group of System Group either directly or indirectly hence System Group acts as root for all Thread Groups in Java.
- System Group contains several system level threads like

Finalizer  
Reference Handler  
Signal Dispatcher  
Attach Listener

(Finalizer (Garbage Collector)  
Yahoo messenger automatically)



Exe class Test

```

    {
        public static void main(String[] args)
        {
            System.out.println(Thread.currentThread().getThreadGroup().getName()); // main
            System.out.println(Thread.currentThread().getThreadGroup().getParent().getName()); // up system
        }
    }
  
```

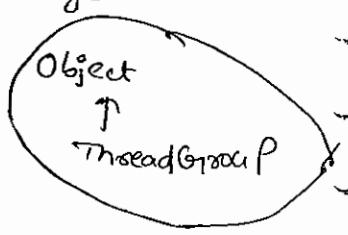
main thread

main thread group

System thread group

System

`ThreadGroup` is a java class present in `java.lang` package  
and it is the direct child class of `Object`.



### Constructors

(1)

```
ThreadGroup g = new ThreadGroup(String groupName);
```

Creates a new `ThreadGroup` with the specified Group name.

The parent of this new Group is the `ThreadGroup` or currently executing thread.

Ex:

```
ThreadGroup g = new ThreadGroup("FirstGroup");
```

(2)

```
ThreadGroup g = new ThreadGroup(ThreadGroup pg, String Groupname);
```

Creates a new `ThreadGroup` with the specified Group name.

The parent of this new `ThreadGroup` is Specified parent group.

Ex:

```
ThreadGroup g1 = new ThreadGroup(g, "SecondGroup");
```

Ex:

```
class Test
```

```
d
```

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

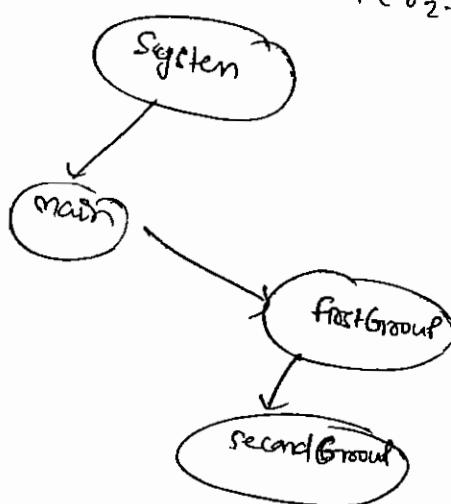
```
{
```

```
    ThreadGroup g1 = new ThreadGroup("First Group");
```

```
    System.out.println(g1.getParent().getName()); // Main
```

```
    ThreadGroup g2 = new ThreadGroup(g1, "Second Group");
```

```
    System.out.println(g2.getParent().getName()); // FirstGroup
```



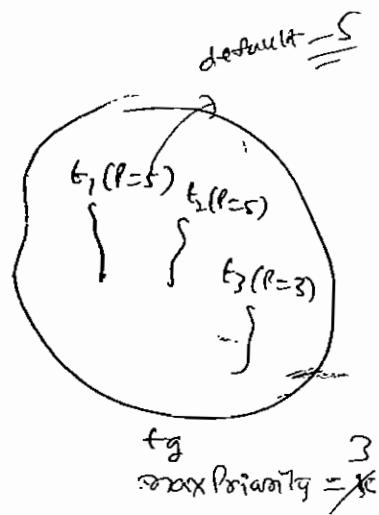
## Important methods of ThreadGroup class

- (1) `String getName()`  
returns name of the ThreadGroup
- (2) `int getMaxPriority()`  
Returns max Priority of ThreadGroup
- (3) `void setMaxPriority(int p)`  
To set maximum Priority of ThreadGroup  
The default max Priority is 10 (ten)

Threads in the ThreadGroup that ~~have~~ already have higher Priority won't be effected but for newly added threads this max Priority is applicable

Ex:

```
==> Class ThreadGroupDemo2
    {
        <> void main(String[] args)
        {
            ThreadGroup g1 = new ThreadGroup("tg");
            Thread t1 = new Thread(g1, "Thread1");
            Thread t2 = new Thread(g1, "Thread2");
            g1.setMaxPriority(3);
            Thread t3 = new Thread(g1, "Thread3");
            System.out.println(t1.getPriority()); 5
            System.out.println(t2.getPriority()); 5
            System.out.println(t3.getPriority()); 3
        }
    }
```



- (4) `ThreadGroup getParent()`

Returns ParentGroup of current thread

- (5) `void list()`

It prints information about ThreadGroup to the console

- (6) `int activeCount()`

Returns number of active threads present in the ThreadGroup

- (7) `int activeGroupCount()`

It returns number of active groups present in the current ThreadGroup

- (8) `int enumerate(Thread[] t)`

To copy all active threads of this ThreadGroup into provided `Thread[]` (Thread array). In this case subThread group threads also will be considered.



① get a name for it

The getMaximumFidelity()

World safety and prosperity (part 2)  
Old set of values  
The default rule is to go to war  
in case of aggression

Threads can be threaded directly difficult ~~but~~ already have higher priority work to be effected but for ready added threads this max priority is applicable

ThreadGroupDemo2  
 Class  
 < max(stealing[0])  
 >  
 < max(stolen[0])>  
 < max(stolen[1])>  
 < max(stolen[2])>

S : (( ( ( getleft() dog  
S : (( ( ( getleft() dog  
Thread max=3 thread  
S : (( ( ( getleft() dog

ThreadGroup getLeave()

void f()

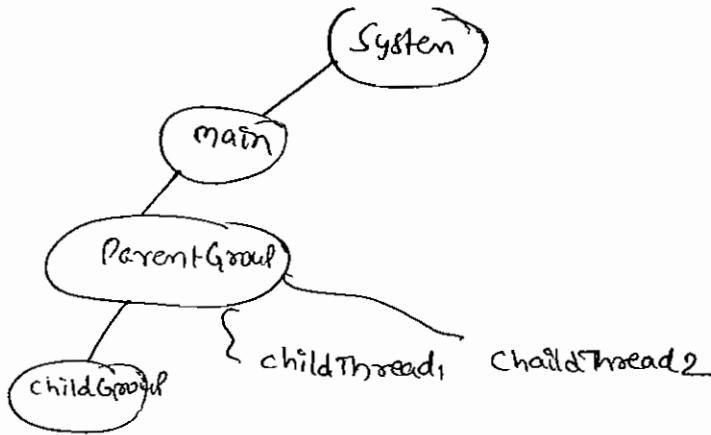
اچیومنٹ (C) میں

active group

• Returns number of active threads present in the threadpool  
• Returns number of active threads present in the threadpool  
• Returns number of active threads present in the threadpool

Retirees present large up-front costs to current retirees.

The readgroup getframe( $\gamma$ )



- Q) write a program to display all active Thread names belongs to System Group and its child Groups

```

class ThreadGroupDemo {
    public static void main(String[] args) {
        ThreadGroup system = Thread.currentThread().getThreadGroup();
        system = system.getParent();
        Thread[] t = new Thread[system.activeCount()];
        system.enumerate(t);
        for (Thread t1 : t) {
            System.out.println(t1.getName() + " - " + t1.isDaemon());
        }
    }
}
  
```

Output  
Reference Handler --- true  
Finalizer --- true  
Signal Dispatcher --- true  
Attach Listener --- true  
main --- false

## java.util.concurrent package

The Problems with traditional synchronized keyword are

- (1) We are not having any flexibility to try for a lock without waiting.
- (2) There is no way to specify maximum waiting time for a thread to get lock so that thread will wait until getting the lock which may creates performance problems which may cause deadlock.
- (3) If a thread releases lock then which waiting thread will get that lock we are not having any control on this.
- (4) There is no API to restart all waiting threads for a lock.
- (5) The synchronized keyword compulsory we have to use either at method level or within the method and it is not possible to use across multiple methods.

To overcome these problems Sun people introduced java.util.concurrent.locks package in 1.5V

It also provides several enhancements to the programmer to provide more control on concurrency.

### ① Lock (I) Interface

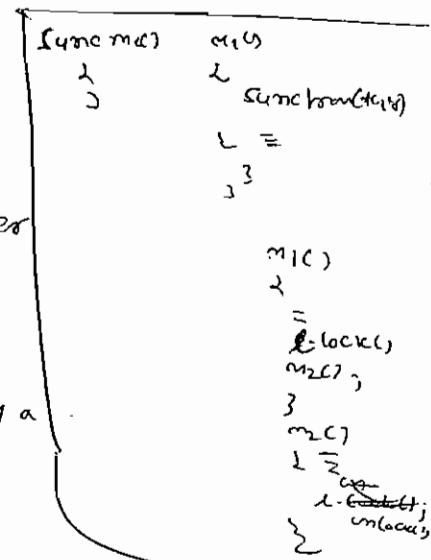
Lock object is similar to implicit lock acquired by a thread to execute synchronized method or synchronized block.

lock implementations provide more extensive operations than traditional implicit locks.

#### Important methods of Lock Interface

##### ① void lock()

If we can use this method to acquire a lock. If lock is already available then immediately current thread will get that lock. If the lock is not already available then it will wait until getting the lock. It is exactly same behaviour as traditional synchronized keyword.



## ② boolean tryLock()

To acquire the lock without waiting

If the lock is available then the thread acquires that lock and returns true. If the lock is not available then this method returns false and can continue its execution without waiting. In this case thread never be entered into waiting state.

```
if (l.tryLock())
{
    Perform Safe operation
}
else {
    Perform alternative operation
}
```

## ③ boolean tryLock(long time, TimeUnit unit)

If lock is available then the thread will get the lock and continue its execution.  
If the lock is not available then the thread will wait until specified amount of time. Still if the lock is not available then thread can continue its execution.

TimeUnit:

TimeUnit is an enum present in `java.util.concurrent.TimeUnit`

Java.util.concurrent  
TimeUnit  
Package

```
enum TimeUnit {
    NANSECONDS,
    MICROSECONDS,
    MILLISSECONDS,
    SECONDS,
    MINUTES,
    HOURS,
    DAYS;
}
```

Ex:

```
if (l.tryLock(1000, TimeUnit.MILLISECONDS))
```

## ④ void lockInterruptibly()

Acquires the lock if it is available and returns immediately if the lock is not available then it will wait.  
While waiting if the thread is interrupted then thread won't get the lock.

## ⑤ void unlock()

To releases the lock.

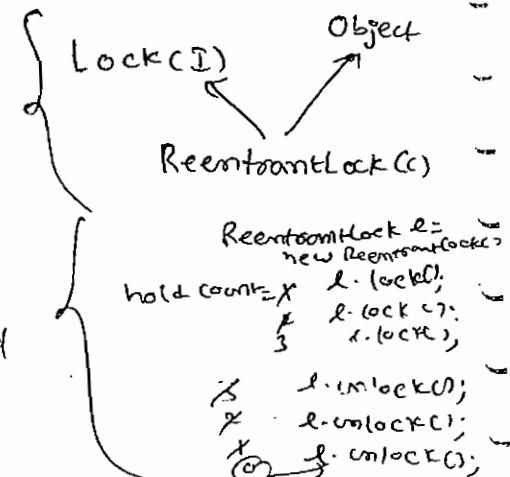
To call this method compulsorily current thread should be owner of the lock otherwise we will get Runtime exception saying ~~IllegalMonitorStateException~~ IllegalMonitorStateException.

## ReentrantLock (c) :

- It is the implementation class of Lock (I) interface and it is the direct child class of Object.

Reentrant means a thread can acquire same lock multiple times without any issue

internally ReentrantLock increments threads personal count whenever we call lock method and decrements count value whenever thread calls unlock() method and lock will be released whenever count reaches zero.



## Constructors

①

```
ReentrantLock l = new ReentrantLock();
```

→ Creates an instance of ReentrantLock

②

```
ReentrantLock l = new ReentrantLock(boolean fairness);
```

Creates ReentrantLock with the given fairness policy

→ if the fairness is true then longest ~~longer~~ waiting thread can acquire the lock (get the lock) if it is available.  
i.e. it follows first come first serve policy (FCFS)

If fairness is false then which waiting thread will get the chance we can't expect.

## Note:

The default value for fairness is false.

- Q) Which of the following declarations are equal?

①

```
ReentrantLock l = new ReentrantLock();
```

②

```
ReentrantLock l = new ReentrantLock(true);
```

③

```
ReentrantLock l = new ReentrantLock(false);
```

All the above

① & ③

① == ③

## Important method of ReentrantLock()

Lock(2)



ReentrantLock(2)

- ① void lock()
- ② boolean tryLock()
- ③ boolean tryLock(long l, TimeUnit t)
- ④ void lockInterruptibly()
- ⑤ void unlock()
- ⑥ int getHoldCount()

Returns number of holds on this lock by current thread

- ⑦ boolean isHeldByCurrentThread()

Returns true iff (if and only if) lock is held by current thread

- ⑧ int getQueueLength()

Returns number of threads waiting for the lock.

- ⑨ Collection getQueuedThreads()

It returns a collection of threads which are waiting to get the lock

- ⑩ boolean hasQueuedThreads()

Returns true if any thread waiting to get the lock

- ⑪ boolean isLocked()

Returns true if the lock is acquired by some thread

- ⑫ boolean isFair()

Returns true if the fairness policy is set with true value

- ⑬ Thread getOwner()

Returns the Thread which acquires the lock

Ex 18

```

import java.util.concurrent.locks.*;
class ReentrantLock2 {
    public static void main(String[] args) {
        ReentrantLock l = new ReentrantLock();
        l.lock();
        l.lock();
        System.out.println(l.isLocked()); // true
        System.out.println(l.isHeldByCurrentThread()); // true
        System.out.println(l.getQueueLength()); // 0
        l.unlock();
        System.out.println(l.getHoldCount()); // 1
        System.out.println(l.isLocked()); // false
        System.out.println(l.isFair()); // false
    }
}

```

Rough Sto

① With synchronized

Class Display

```

public void wish(String name) {
    for (int i=0; i<10; i++) {
        synchronized {
            System.out.println("Good morning");
        }
        Thread.sleep(2000);
    }
    System.out.println("Good night");
}

```

Display d=new Display();

d.wish("Dhoni");

Good morning: Dhoni

"

Action not responsible to  
call fair method my thread  
to this job how?

Class myThread extends Thread

Display d,

String name;

myThread (Display d, String name)

{ this=d;

this.name=name;

② Public void anal

{ d.wish(name);

③ Class SynchronizedDemo

```

public static void main(String[] args) {
    Display d=new Display();
    MyThread t1=new MyThread(d, "Dhoni");
    MyThread t2=new MyThread(d, "Yuvraj");
    t1.start();
}

```

new MyThread (Display d, String name)

{ this=d;

this.name=name;

wish(name);

If I want at a time only one method execute. So then use synchronized with out declaring with() method as synchronized use ReentrantLock to get the o/p (SynchronizedDemo.java)

→ traditional Synchronization  
not have try (lock();)

Ex 28

```
import java.util.concurrent.locks.*;
```

```
class Display
```

{

```
    ReentrantLock l = new ReentrantLock();
```

```
    public void wish (String name)
```

{

l.lock();

→ ①

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

{

```
            System.out.println ("Good Morning");
```

}

Thread.sleep (2000);

}

```
    catch (InterruptedException e)
```

{

}

}

l.unlock();

→ ②

```
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);
```

}

```
class ReentrantLockDemo
```

{

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

{

```
        Display d = new Display();
```

```
        MyThread t1 = new MyThread (d, "Dhoni");
```

```
        MyThread t2 = new MyThread (d, "Yuvraj");
```

```
        MyThread t3 = new MyThread (d, "Kohli");
```

t1.start();

t2.start();

t3.start();

If we comment line ① and line ② simultaneously and we will get irregular output

If we are not commenting lines ① & ② then the threads will be executed one by one and we will get Regular output.

Demo program for tryLock() method

```

Ex 8 import java.util.concurrent.locks.*;
class MyThread extends Thread
{
    static ReentrantLock l = new ReentrantLock();
    MyThread (String name)
    {
        super(name);
    }
    public void run()
    {
        if (l.tryLock())
        {
            System.out.println(Thread.currentThread().getName() + "... got lock"
                + " and performing safe operations");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e) {}
            l.unlock();
        }
        else
        {
            System.out.println(Thread.currentThread().getName() + "... "
                + "unable to get lock and performing alternative"
                + "operations");
        }
    }
}
class ReentrantLockDemo3
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread("First Thread");
        MyThread t2 = new MyThread("Second Thread");
        t1.start();
        t2.start();
    }
}

```

if (l.tryLock())
   
 {
 else
 Safe operations
 }
 else
 {
 = Alternative
 "Operations"
 }
 }
 Sometimes may
 be second thread
 also set the lock

O/P:

- first Thread... get lock and performing safe operations  
 second thread... unable to get lock and hence performing alternative operations

Expt

```

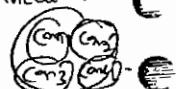
import java.util.concurrent.locks.*;
import java.util.concurrent.*;
class Mythread extends Thread {
    static ReentrantLock l = new ReentrantLock();
    super(name);
    public void run() {
        do {
            try {
                if (tryLock(5000, TimeUnit.MILLISECONDS))
                    System.out.println(Thread.currentThread().getName() + " got lock");
                Thread.sleep(30000);
                l.unlock();
            } catch (Exception e) {}
        } while (true);
    }
    class ReentrantLockDemo {
        static void main(String[] args) {
            Mythread t1 = new Mythread("first thread");
            Mythread t2 = new Mythread("second thread");
            t1.start();
            t2.start();
        }
    }
}
  
```

until getting lock  
 keep on trying  
 after 30sec  
 second thread releases lock

off  
 first thread ... got lock  
 second thread --- unable to get lock and will try again  
 second thread --- unable to get lock and will try again  
 second thread --- unable to get lock and will try again  
 second thread --- unable to get lock and will try again  
 second thread --- unable to get lock and will try again  
 second thread --- unable to get lock and will try again  
 first thread ... releases lock  
 second thread --- got lock  
 second thread --- releases lock.

## Thread Pools (Executor framework)

like  
connection pool



Creating a new thread for every job may create performance and memory problems to overcome this we should go for Thread Pool.

Thread Pool is a pool of already created threads ready to do our job.

→ java 1.5 version introduces Thread Pool framework to implement Thread Pools

→ Thread Pool framework also known as Executor framework

→ we can create a Thread Pool as follows

`ExecutorService service = Executors.newFixedThreadPool(3);`

→ we can submit a Runnable job by using submit() method

`service.submit(job);`

→ we can shutdown ExecutorService by using shutdown() method.

`service.shutdown();`

3.13

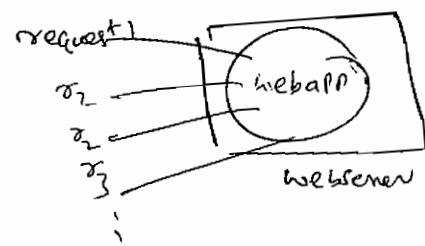
A group of  
 Runnable objects  
 Created  
 PrintJobs → Runnable  
 obj  
 MyRunnable  
 = new MyRunnab  
 le();  
 Thread t = new Thread  
 (m);  
 6 time we do

```

Exe import java.util.concurrent.*;
class PrintJob implements Runnable
{
  String name;
  PrintJob(String name)
  {
    this.name = name;
  }
  public void run()
  {
    System.out.println("Job Started by Thread " + Thread.currentThread().getName());
    try
    {
      Thread.sleep(5000);
    }
    catch (InterruptedException e)
    {
    }
    System.out.println("Job completed by Thread " + Thread.currentThread().getName());
  }
}
class ExecutorDemo
{
  public static void main(String[] args)
  {
    PrintJob[] jobs = {
      new PrintJob("durga"),
      new PrintJob("Ravi"),
      new PrintJob("Shiva"),
      new PrintJob("Pavan"),
      new PrintJob("Suresh"),
      new PrintJob("Anil")
    };
    ExecutorService service = Executors.newFixedThreadPool(3);
    for (PrintJob job : jobs)
    {
      service.submit(job);
    }
    service.shutdown();
  }
}
  
```

In the above example ③ threads are responsible to execute ⑥ jobs so that a single thread can be reused for multiple jobs

notes  
while designing web servers and application servers we can use Threadpool concept.



## Callable and Future

- (i) In the case of Runnable job thread won't return any thing after completing the job.
- (ii) If a thread is required to return some result after execution then we should go for Callable
- (iii) Callable interface contains only one method call().

Public Object call() throws Exception

- (iv) If we submit Callable object to Executors then after completing the job thread returns an object of the type Future.  
i.e. Future object can be used to retrieve the result from Callable job.

Ex:

```
import java.util.concurrent.*;
class MyCallable implements Callable
{
    int num;
    MyCallable(int num)
    {
        this.num = num;
    }
}
```

Class myRunnable implements Runnable  
{  
 public void run()  
 {  
 try  
 {  
 catch()  
 }  
 }  
}  
myRunnable r=new myRunnable();  
MyThread t=new MyThread(r);  
here  
MyThread job  
MyRunnable callable not  
return anything void

public Object call() throws Exception

d

Sop (Thread.currentThread().getName() + " is ...")

responsible to find sum of first " + num " numbers") ;

int sum = 0;

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

d

sum = sum + i;

y

return sum;

}

class CallableFutureDemo

d

public static void main(String[] args) throws Exception

d

MyCallable[] jobs = { new MyCallable(10),

new MyCallable(20),

new MyCallable(30),

new MyCallable(40),

new MyCallable(50),

new MyCallable(60) } ;

ExecutorService service = Executors.newFixedThreadPool(3);

for (MyCallable job : jobs)

d

Future f = service.submit(job);

y Sop (f.get());

service.shutdown();

3}

O/P:

55	-	-
210	-	-
465	-	-
820	-	-
1275	-	-
1830	-	-

## Differences b/w Runnable and Callable

Runnable	Callable
① If a thread is not required to return anything after completing the job then we should go for Runnable	① If a thread required to return something after completing the job then we should go for Callable
② Runnable Interface contains only one method run()	② Callable Interface contains only one method call()
③ Runnable job is not required to return anything and hence return type of run() method is void	③ Callable job is required to return something and hence return type of call() method is Object
④ With in the run() method if there is any chance of raising checked exception compulsorily we should handle by using try-catch because we can't use throws keyword for run()	④ <del>throws</del> within call() method if there is any chance of raising checked exception we are not required to handle by using try-catch because call() method already throws exception
⑤ Runnable interface present in java.lang package	⑤ Callable interface present in java.util.concurrent package
⑥ Introduced in 1.0 version	⑥ Introduced in 1.5 version

Rough

Runnable

```
d & void run();
```

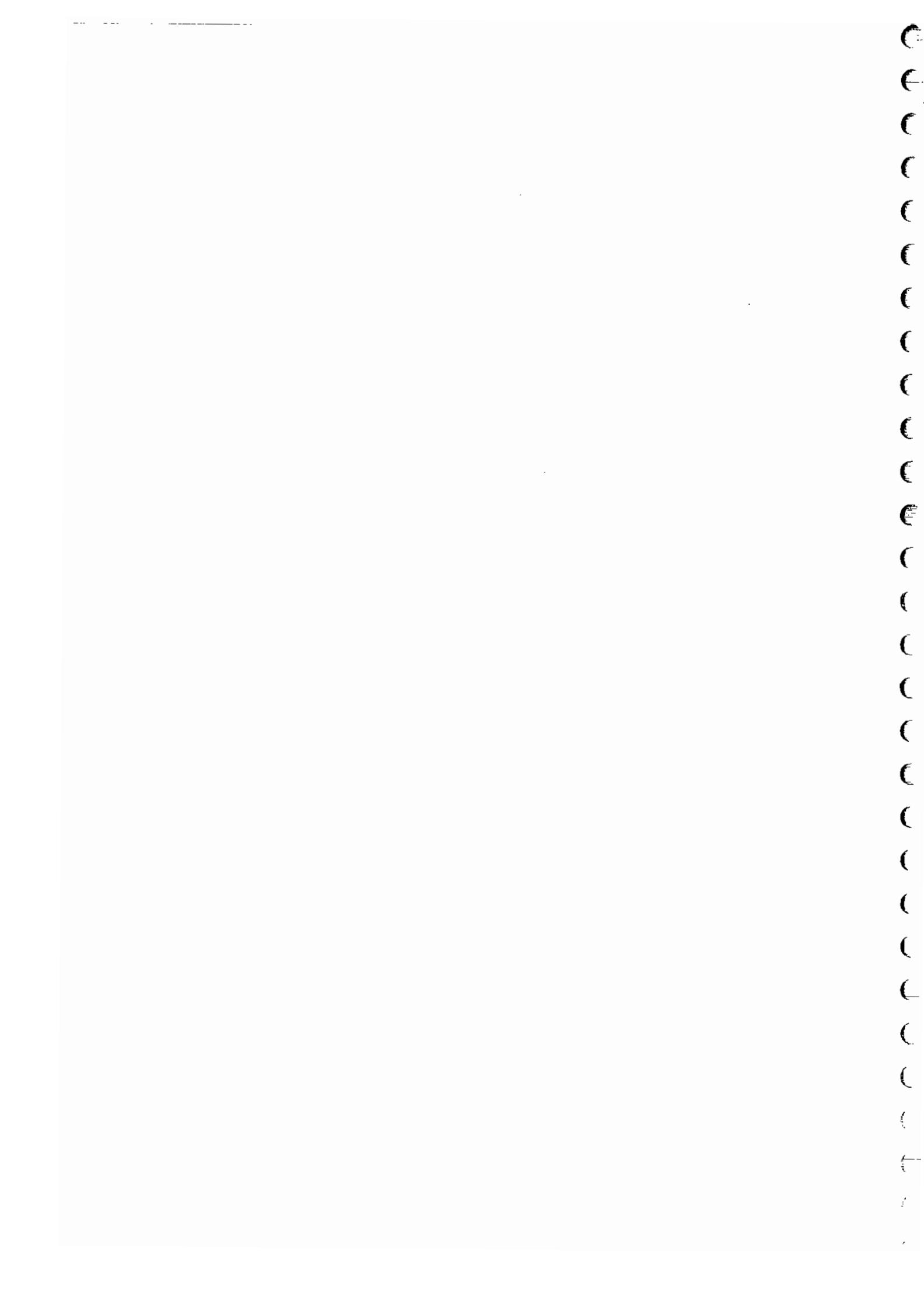
}

if run() throws exception  
while implementing we can't take  
this

because class doesn't throw any  
exception

so use catch

卷之三



## Thread Local

ThreadLocal class provides ThreadLocal variables.

ThreadLocal class maintains values per Thread basis

each ThreadLocal Object maintains a separate value like user id, transaction id etc for each Thread that accesses that object.

Thread can access its local value, can manipulate its value and even can remove its value.

In every part of the code which is executed by the Thread we can access its local variable.

Ex: Consider a service which invokes some business methods we have requirement to generate a unique transactionid for each and every request and we have to pass this transactionid to the business methods for this requirement we can use ThreadLocal to maintain a separate transactionid for every request i.e. for every Thread.

### Notes

ThreadLocal class introduced in 1.2 version and enhanced in 1.5 version for ThreadLocal can be associated with ThreadScope.

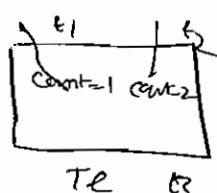
Total code which is executed by the thread has access to the corresponding ThreadLocal variables.

- A Thread can access its own local variables and can't access other threads local variables.

- Once Thread entered into dead state all its local variables are by default ~~eligible~~ for Garbage collection

### Constructor

① ThreadLocal tl = new ThreadLocal();  
Creates a ThreadLocal variable



### Methods

① **Object get()**

Returns the value of ThreadLocal variable associated with current Thread

② **Object initialValue()**

Returns initial value of ThreadLocal variable associated with current thread.

The default implementation of this method returns null.

To customize our own initial value we have to override this method

③ void set(Object newValue)

To set a new value

④ void remove()

To remove the value of ThreadLocal variable associated with current thread.

- It is newly added method in 1.5 version

- After removal if we are trying to access it will be reinitialized once again by invoking its initialValue() method.

Ex1

```
class ThreadLocalDemo1  
{  
    public static void main(String[] args)  
    {  
        ThreadLocal tl=new ThreadLocal();  
        System.out.println(tl.get()); // null  
        tl.set("durga");  
        System.out.println(tl.get()); // durga  
        tl.remove();  
        System.out.println(tl.get()); // null  
    }  
}
```

OVERRIDING OF initialValue() method

```
class ThreadLocalDemo1A  
{  
    public static void main(String[] args)  
    {  
        ThreadLocal tl=new ThreadLocal()  
        {  
            public Object initialValue()  
            {  
                return "abc";  
            }  
        };  
        System.out.println(tl.get()); // abc  
        tl.set("durga");  
        System.out.println(tl.get()); // durga  
        tl.remove();  
        System.out.println(tl.get()); // abc  
    }  
}
```

Ex: 8

```

class CustomerThread extends Thread
{
    static Integer custId = 0;
    private static ThreadLocal tl = new ThreadLocal();
    protected Integer initialValue()
    {
        return ++custId;
    }
    CustomerThread (String name)
    {
        super(name);
    }
    public void run()
    {
        System.out.println(Thread.currentThread().getName() + " executing with "
                           + "Customer Id :" + tl.get());
    }
}

```

```
class ThreadLocalDemo2
```

```
{
```

```

    public void main (String[] args)
    {
        CustomerThread c1 = new CustomerThread ("Customer Thread -1");
        CustomerThread c2 = new CustomerThread ("Customer Thread -2");
        CustomerThread c3 = new CustomerThread ("Customer Thread -3");
        CustomerThread c4 = new CustomerThread ("Customer Thread -4");
        c1.start();
        c2.start();
        c3.start();
        c4.start();
    }
}
```

In the above program for every customer thread a separate customer id will be maintained by ThreadLocal object.

### ThreadLocal Vs Inheritance

Parent threads ThreadLocal variable by default not available to the child thread. If we want to make Parent threads ThreadLocal variable value available to the child thread then we should go for InheritableThreadLocal class.

- By default child threads value is exactly same as ParentThread's value but we can provide customized value for child thread by overloading ~~childValue~~ childValue() method.

## Constructor

InheritableThreadLocal tl = new InheritableThreadLocal();

## methods

InheritableThreadLocal is the child class of ThreadLocal and hence all methods present in ThreadLocal by default available to InheritableThreadLocal.

In addition to these methods it contains only one method

Public Object childValue(Object parentValue)

## Ex:

```

class ParentThread extends Thread
{
    public static InheritableThreadLocal tl = new InheritableThreadLocal()
    {
        public Object childValue (Object p)
        {
            return "cc";
        }
    }
    public void run()
    {
        tl.set("pp");
        System.out.println("Parent Thread value---" + tl.get());
        ChildThread ct = new ChildThread();
        ct.start();
    }
}

class ChildThread extends Thread
{
    public void run()
    {
        System.out.println("Child Thread value---" + parentThread.tl.get());
    }
}

class ThreadLocalDemo3
{
    public static void main (String [] args)
    {
        ParentThread pt = new ParentThread();
        pt.start();
    }
}

```

O/P:

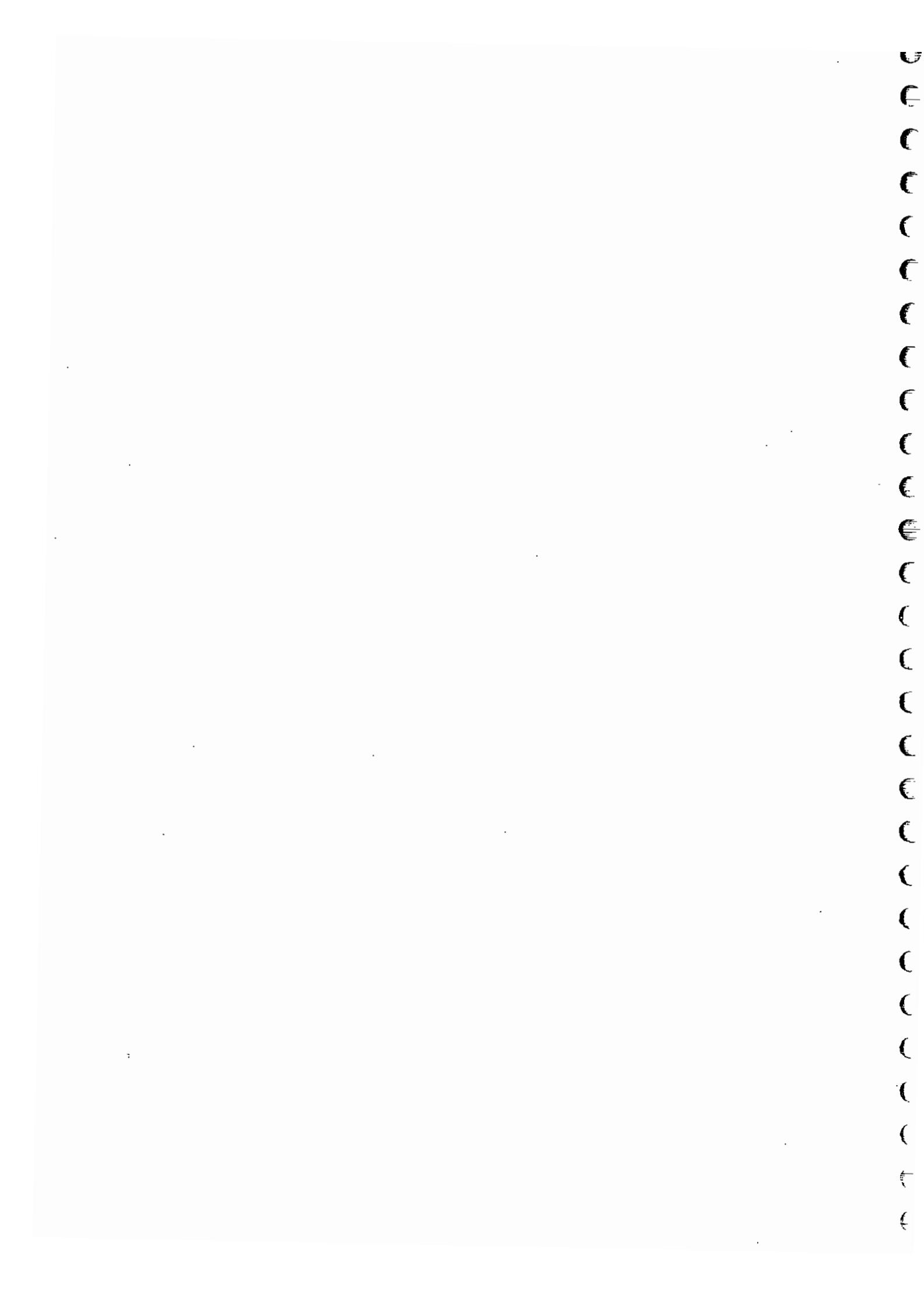
Parent Thread value -- pp
Child Thread value -- cc

- In the above program if we replace InheritableThreadLocal with ThreadLocal and overriding childValue() method then the o/p is
 

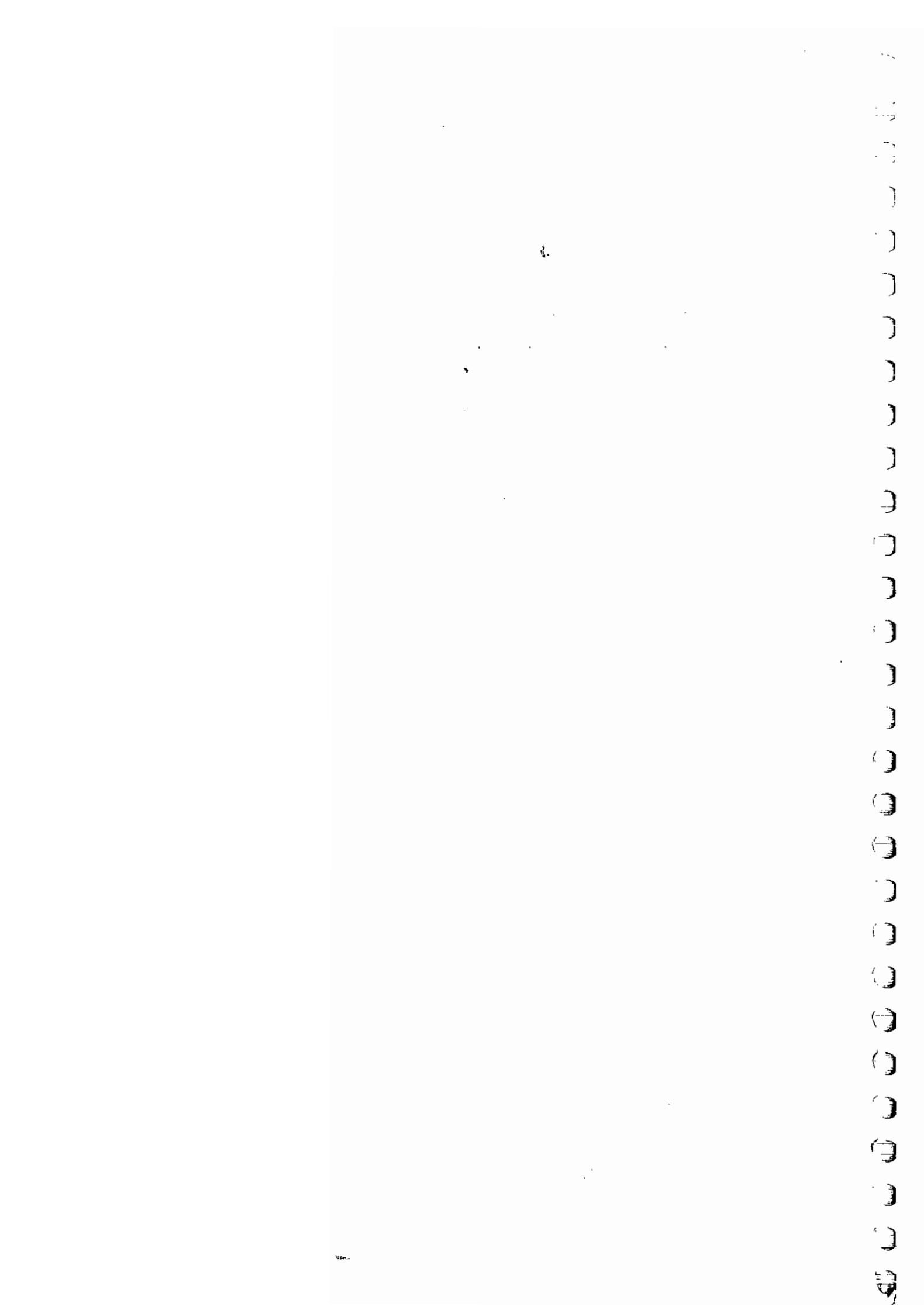
Parent Thread value -- pp
---------------------------

In the above program if we are maintaining inheritable ThreadLocal  
and if we are not overriding childValue() method then the output is

Parent Thread value -- pp  
Child Thread value --- pp



10. java.lang package 1- 50
11. java.io package 51- 66
12. Serialization 67- 85
13. Collection framework 87- 146
14. Generics 147- 162
15. Inner classes 163- ~~184~~
16. Internationalization (I18N) ~~185~~ 185- 192
17. Development 193- 205
18. Regular expressions 207- 216
19. enum 217- 228
20. JVM Architecture 227- 250



## java.lang Package

- ① Introduction
- ② Object class
- ③ String class
- ④ StringBuffer class
- ⑤ StringBuilder class
- ⑥ Wrapper classes
- ⑦ AutoBoxing & AutoUnBoxing

### ① Introduction

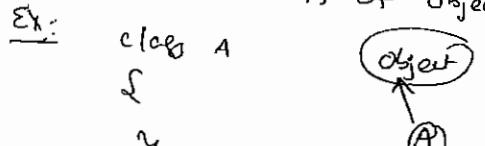
for writing any Java Program whether it is simple or complex the most commonly required classes and interfaces are grouped onto a separate package which is nothing but java.lang package.  
we are not required to import java.lang package explicitly because all classes and interfaces present in lang package by default available in every Java program.

### ② java.lang.Object

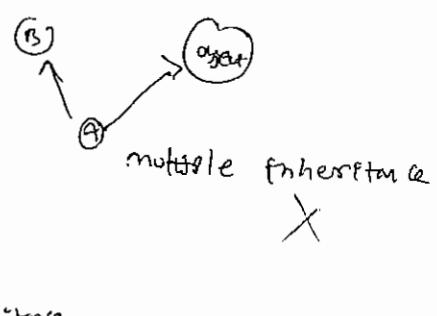
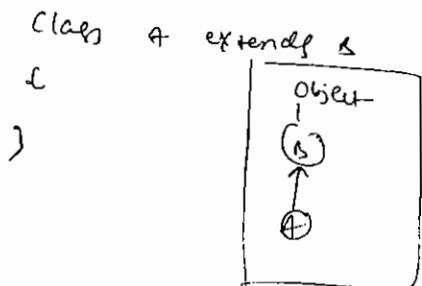
The most commonly required methods for every Java class (whether it is predefined class or customized class) are defined in a separate class which is nothing but Object class.  
 • Every class in Java is the child class of Object either directly or indirectly.  
 • Hence Object class is considered as root of all Java programs.

#### Note:

- ① If our class doesn't extend any other class then only our class is the direct child class of Object.



- ② If our class extends any other class then our class is an indirect child class of Object.



## Conclusion

- either directly or indirectly Java won't provide support for multiple inheritance with respect to classes.

Object class defines the following 12 methods

- (1) public String toString()
- (2) public native int hashCode()
- (3) public boolean equals(Object o)
- (4) protected native Object clone() throws CloneNotSupportedException
- (5) protected void finalize() throws Throwable
- (6) public final Class getClass()
- (7) public final void wait() throws InterruptedException
- (8) public final void wait(long ms) throws InterruptedException
- (9) public final void wait(long ms, int ns) throws InterruptedException
- (10) public native final void notify()
- (11) public native final void notifyAll()

## Note

Strictly speaking Object class contains 12 methods the extra method is registerNatives().

Private static native void registerNatives();

This method internally required for Object class and not available to the child classes. hence we are not required to consider this method.

- (1) public String toString()
- (2) toString()

we can use toString() method to get string representation of an object.

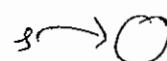
String s=(obj).toString();

whenever we are trying to print object reference internally toString() method will be called.

Student

s = new Student();

System.out.println(s);  $\Rightarrow$  System.out.println(s.toString());



if our class doesn't contain `toString()` method then Object class `toString()` method will be executed.

Ex: Class Student

```
class Student {
    String name;
    int rollno;
    Student(String name, int rollno) {
        this.name = name;
        this.rollno = rollno;
    }
    public static void main(String[] args) {
        Student s1 = new Student("Durga", 101);
        Student s2 = new Student("Ravi", 102);
        System.out.println(s1); // Output: Student@1e88759
        System.out.println(s1.toString()); // Student@1e88759
        System.out.println(s2); // student@6e1408
    }
}
```

$s_1 \rightarrow \text{Durga}$   
 $s_2 \rightarrow \text{Ravi}$

In the above example Object class `toString()` method got executed which is implemented as follows.

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(
        hashCode());
}
```

ClassName @ hashCode-in-hexadecimal-form

Based on our requirement we can override `toString()` method to provide our own string representation.

for example whenever we are trying to print student object reference to print his name and rollno we have to override `toString()` method as follows

Ex:

```
public String toString() {
    return name + " -- " + rollno;
    // return "This is student with the name : " + name + " and
    // Rollno : " + rollno;
}
```

→ In all wrapper classes, in all collection classes, string class, StringBuffer & StringBuilder classes toString() method is overridden for meaningful string representation. Hence it is highly recommended to override toString() method in our class also.

Ex:

```
class Test
{
    public String toString()
    {
        return "test";
    }

    public static void main(String[] args)
    {
        String s = new String("durga");
        System.out.println(s); // durga

        Integer i = new Integer(10);
        System.out.println(i); // 10

        ArrayList l = new ArrayList();
        l.add("A");
        l.add("B");
        System.out.println(l); // [A, B]

        Test t = new Test();
        System.out.println(t); // test
    }
}
```

02-09-14

②

## hashCode()

- for every object a unique number generated by JVM which is nothing but hashCode.
- hashCode won't represents address of Object
- JVM will use hashCode while saving objects into hashing related datastructures like hashtable, hashmap, hashset etc...
- The main advantage of saving objects based on hashCode is search operation will become easy (the most powerful search algorithm upto today is hashing).
- If we are giving the chance to Object class hashCode() method it will generates hashCode based on address of the object it doesn't mean hashCode represents address of the object.
- Based on our requirement we can override hashCode() method in our class to generate our own hashCode.

Overriding hashCode() method is said to be proper iff (if and only if)  
for every object we have to generate a unique member of hashCode.

1) class Student

```
    {
        public int hashCode()
        {
            return 100;
        }
    }
```

*(Improper way)*

This is Improper way of overriding hashCode() method because for all Student objects we are generating same member as hashCode.

2) class Student

```
    {
        public int hashCode()
        {
            return rollno;
        }
    }
```

*(Proper way)*

This is Proper way of overriding hashCode method because we are generating a different member as hashCode for every object.

## toString() vs hashCode()

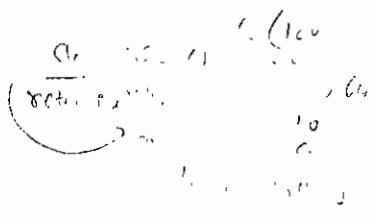
If we are giving the chance to object class toString() method it will internally calls hashCode() method.

If we are overriding toString() method then our toString() method may not call hashCode() method.

Ex 16

```
class Test
{
    int i;
    Test(int i)
    {
        this.i = i;
    }
    public static void main(String[] args)
    {
        Test t1 = new Test(10);
        Test t2 = new Test(100);
        System.out.println(t1); Test @axd
        System.out.println(t2); Test @bde
        Object o = t1.toString();
        Object o2 = t2.hashCode();
    }
}
```

```
① public String toString()
{
    return "Object "+i+" "+this.hashCode();
}
② public int hashCode()
{
    return i;
}
```



class Test

```
 { int i;
  Test(int i)
  {
    this.i = i;
  }
  public int hashCode()
  {
    return i;
  }
  ~main(String[] args)
  {
    Test t1 = new Test(10);
    Test t2 = new Test(100);
    System.out.println("t1 = " + t1);
    System.out.println("t2 = " + t2);
    System.out.println("t1 = " + t1);
    System.out.println("t2 = " + t2);
    System.out.println("t1 = " + t1);
    System.out.println("t2 = " + t2);
  }
}
```

class Test

```
 { int i;
  Test(int i)
  {
    this.i = i;
  }
  public String toString()
  {
    return i + " ";
  }
  public int hashCode()
  {
    return i;
  }
  public static void main(String[] args)
  {
    Test t1 = new Test(10);
    Test t2 = new Test(100);
    System.out.println(t1);
    System.out.println(t2);
    System.out.println(t1);
    System.out.println(t2);
    System.out.println(t1);
    System.out.println(t2);
  }
}
```

Test → toString()

### ③ equals()

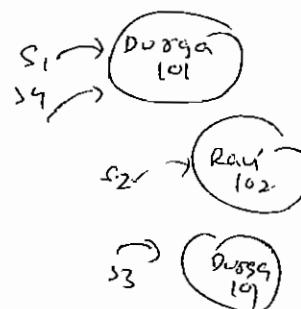
We can use equals() method to check equality of two objects

Ex: `obj1.equals(obj2)`

If our class doesn't contain equals() method then Object class's equals() method will be executed.

Ex:

```
class Student
{
  String name;
  int rollno;
  Student(String name, int rollno)
  {
    this.name = name;
    this.rollno = rollno;
  }
  ~main(String[] args)
  {
    Student s1 = new Student("Durga", 101);
    Student s2 = new Student("Ravi", 102);
    Student s3 = new Student("Durga", 101);
    Student s4 = s1;
    System.out.println(s1.equals(s2)); // false
    System.out.println(s1.equals(s3)); // false
    System.out.println(s1.equals(s4)); // true
  }
}
```



for the above example object class equals() method got executed which is meant for reference comparison (address comparison) i.e if two references pointing to the same object then only equals() method returning true. (7)

→ based on our requirement we can override equals() method for Content comparison.

04-09-14

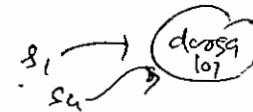
while overriding equals() method for Content comparison we have to take care about the following

- ① what is the meaning of equality (i.e whether we have to check only names or only rollno's or both)
- ② If we are passing different type of object our equals method should not rise ~~ClassCastException~~ exception i.e. we have to handle ~~ClassCastException~~
- ③ If we are passing null argument then our equals() method should return false.

The following is the proper way of overriding equals method for student class content comparison

```
public boolean equals(Object obj){  
    if (obj == this) {  
        return true;  
    }  
    if (obj instanceof Student) {  
        Student s = (Student) obj;  
        if (name.equals(s.name) && rollno == s.rollno) {  
            return true;  
        }  
    }  
    return false;  
}
```

Ex:  
s1.equals("dss")  
s1.equals(null),

Student  $s_1$  = new Student ("durga", 101);   
 student  $s_2$  = new Student ("Ravi", 102);   
 student  $s_3$  = new Student ("durga", 101);   
 student  $s_4$  =  $s_1$ ;  
~~sop (s1.equals(s2)); false~~  
~~sop (s1.equals(s3)); false~~ ~~true~~  
~~sop (s1.equals(s4)); true~~  
~~sop (s1.equals("durga")); 1 false~~  
~~sop (s1.equals(null)); 1 false~~

### Simplified Version of equals() method

```

public boolean equals (Object obj)
{
    try
    {
        Student s = (Student) obj;
        if (name.equals (s.name) && rollno == s.rollno)
            return true;
        else
            return false;
    }
}

```

```

}
    catch (ClassCastException e)
    {
        return false;
    }
    catch (NullPointerException e)
    {
        return false;
    }
}

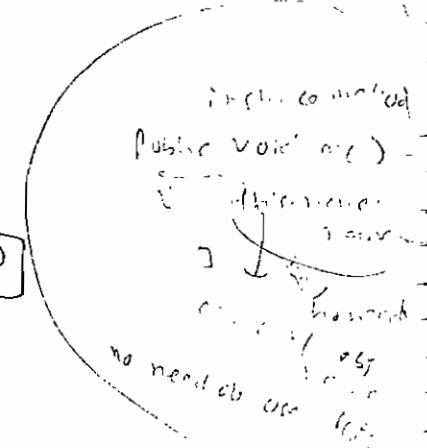
```

### More Simplified Version of equals() method

```

public boolean equals (Object obj)
{
    if (obj instanceof Student)
    {
        Student s = (Student) obj;
        if (name.equals (s.name) && rollno == s.rollno)
            return true;
        else
            return false;
    }
    return false;
}

```



Note: To make above equals methods more efficient we have to write the following code at the beginning inside equals() method.

```
if (obj == this)  
    return true;
```

according to this if both references pointing to the same object then without performing any comparison  
• equals() method returns true directly

SOP(S, m1)(  
e.)

```
String s1 = new String ("durga");  
String s2 = new String ("durga");  
SOP (s1 == s2); false  
SOP (s1.equals(s2)); true
```

In Strong class • equals() method  
is overridden for Content Comparison  
hence, even though objects are different  
if content is same then • equals()  
method returns true

```
StringBuffer sb1 = new StringBuffer  
("durga");  
StringBuffer sb2 = new StringBuffer  
("durga");  
SOP (sb1 == sb2); false  
SOP (sb1.equals(sb2)); false
```

In StringBuffer • equals() method is not  
overridden for Content Comparison  
hence if objects are different  
• equals() method returns false even though  
content is same.

05-09-2014

## Relation b/w == operator and equals() method

① If two objects are equal by == operator then these objects are always equal by .equals() method.

i.e.  $r_1 == r_2$  is true then  $r_1.equals(r_2)$  is true

always

② If two objects are not equal by double equal operator (==) then we can't conclude anything about .equals() method it may return true or false i.e. if  $r_1 == r_2$  is false then  $r_1.equals(r_2)$  may returns true or false so we can't expect exactly.

( $r_1 \neq r_2$ )

③ If two objects are equal by .equals() method then we can't conclude anything about == (double equal) operator. It may returns true or false i.e. if  $r_1.equals(r_2)$  is true then we can't conclude anything about  $r_1 == r_2$  it may returns true or false

④ If two objects are not equal by .equals() method then these objects are not equal by == operator  
i.e. if  $r_1.equals(r_2)$  is false, then  $r_1 == r_2$  is always false

Check  
Content &  
reference

## Differences b/w == operator and .equals() method?

To use == (double equal) operator compulsorily there should be

some relation b/w argument types (either child to parent or parent to child or same type)

otherwise we will get compiletime error saying incomparable types,

→ if there is no relation b/w argument types then .equals() method can't rise any compiletime or runtime errors simply it returns false

Ex. String s<sub>1</sub> = new String ("durga");

String s<sub>2</sub> = new String ("durga");

StringBuffer s<sub>1</sub> = new StringBuffer ("durga");

StringBuffer s<sub>2</sub> = new StringBuffer ("durga");

$sop(s_1 == s_2);$  false

$sop(s_1.equals(s_2));$  true

```

sop (sb1 == sb2); false
sop (sb1.equals (sb2)); false
sop (s1 == sb1); 
sop (s1.equals (sb1)); false

```

Ex: Incompatible types

java.lang.String and  
java.lang.StringBuffer

operator overloading  
is not there in Java  
3)

### $\text{==}$ operator

### `• equals () method`

① It is an operator in Java applicable for both primitives and object types

② In the case of object references  
 $\text{==}$  (double equal operator) meant for reference comparison (Address comparison)

③ We can't override  $\text{==}$  operator for content comparison

④ To use  $\text{==}$  operator  
Compulsory there should be some relation b/w argument types (either child to Parent)  
(or) Parent to child or same type otherwise we will get compilation error saying incompatible types

① It is a method applicable only for object types but not for primitives

② By default `• equals ()` method present in Object class also meant for reference comparison

③ We can override `• equals ()` method for content comparison

④ If there is no relation b/w argument types then `• equals ()` method won't rise any compiletime or runtime errors and simply returns false

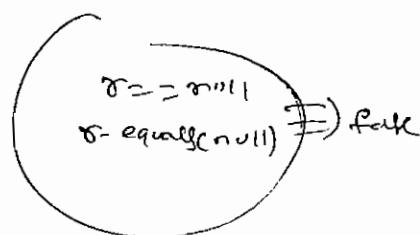
### Answer in one line

In general we can use  $\text{==}$  operator for reference comparison and `• equals ()` method for content comparison.

Note: for any object reference  $\tau$ ,

$\tau == \text{null} \Leftrightarrow \tau.equals (\text{null})$  always returns false;

Ex:



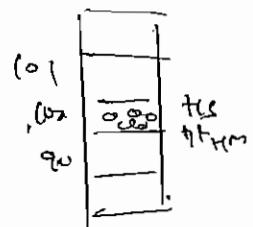
Thread t = new Thread();

sop (t == null); false  
sop (t.equals (null)); false

NOTE Hashing related data structures follow the following fundamental rule.

Two equivalent Objects should be placed in same bucket but all objects present in the same bucket need not be equal.

Contract b/w .equals() method and hashCode()



- ① If two objects are equal by .equals() method then their hashcodes must be equal i.e two equivalent objects should have same hashCode.  
i.e if  $r_1.equals(r_2)$  is true then  $r_1.hashCode() == r_2.hashCode()$  is always true.
- ② Object class .equals() method and hashCode() method follows above contract hence whenever we are overriding .equals() method compulsory we should override hashCode() method to satisfy above contract.  
(i.e Two equivalent objects should have same hashCode)
- ③ If two objects are not equal by .equals() method then there is no restriction on hashCode they may be equal or may not be equal
- ④ If hashCode of two objects are equal then we can't conclude anything about .equals() method it may return true or false
- ⑤ If hashCode of two objects are not equal then these objects are always not equal by .equals() method.

#### Notes

To satisfy contract between .equals() and hashCode() methods whenever we are overriding .equals() method compulsory we have to override hashCode() method otherwise we won't get any compiletime or runtime errors but it is not a good programming practice.

In String class .equals() method is overridden for content comparison and hence hashCode() method is also overridden to generate hashCode based on content.

Eg's

```
String s1 = new String("durga");
String s2 = new String("durga");
System.out.println(s1.equals(s2)); // true
System.out.println(s1.hashCode()); // 9595049
System.out.println(s2.hashCode()); // 9595049
```

it on StringBuffer .equals() method is not overridden for Content Comparison and hence hashCode() method is also not overridden.

Ex

```
StringBuffer sb1 = new StringBuffer("durga");
```

```
StringBuffer sb2 = new StringBuffer("durga");
```

```
sb1.equals(sb2); false
```

```
sb1.hashCode(); 19621457
```

```
sb2.hashCode(); 4872882
```

Q) Consider the following Person class

```
public boolean equals(Object obj){  
    if(obj instanceof Person){  
        Person p = (Person) obj;  
        if(name.equals(p.name) && age == p.age)  
            return true;  
        else  
            return false;  
    }  
}
```

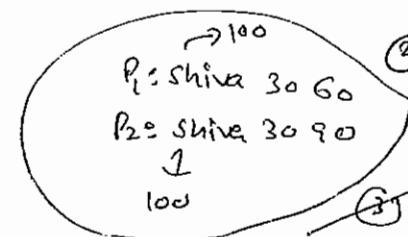
which of the following hashCode methods are appropriate for Person class

① public int hashCode(){  
 return 100;

② public int hashCode(){  
 return age + name;

③ public int hashCode(){  
 return name.hashCode() +  
 age;

④ no restrictions

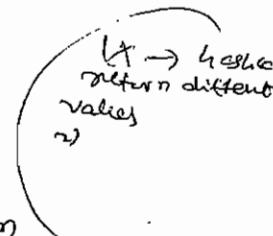


### Notes

Based on which parameters we override .equals() method, it is highly recommended to use same parameters while overriding hashCode method also.

### Notes

On all Collection classes, in all wrapper classes and in String class .equals() method is overridden for Content Comparison hence it is highly recommended to override equals() method in our class also for Content Comparison.



## Q609(14) ④ Clone()

- The process of creating exactly duplicate object is called cloning.
- The main purpose of cloning is to maintain backup copy and to preserve state of an object.
- We can perform cloning by using clone() method of Object class.

protected native Object clone() throws CloneNotSupportedException

Ex:

```
class Test implements Cloneable  
{  
    int i=10;           ② RE  
    int j=20;  
    public void main(String[] args) throws CloneNotSupportedException  
    {  
        Test t1 = new Test();  
        Test t2 = (Test) t1.clone();          ② CE  
        t2.i = 888;  ① CE  
        t2.j = 999;  
        System.out.println(t1.i + " " + t1.j);  
        System.out.println(t2.i + " " + t2.j);  
    }  
}
```

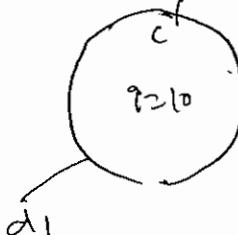
Diagram illustrating the cloning process:

- Object  $t_1$  contains  $i=10$  and  $j=20$ .
- Object  $t_2$  is a clone of  $t_1$ , so it also contains  $i=10$  and  $j=20$ .

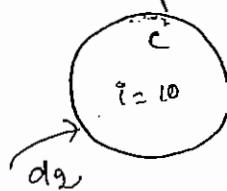
- We can perform cloning only for cloneable objects.
- An object is said to be cloneable if its class implements Cloneable interface.
- Cloneable interface present in java.lang package and it doesn't contain any methods. It is a marker interface.
- If we are trying to perform cloning for non cloneable objects then we will get RuntimeException saying ~~clone~~ CloneNotSupportedException.

Shallow Cloning vs Deep Cloning

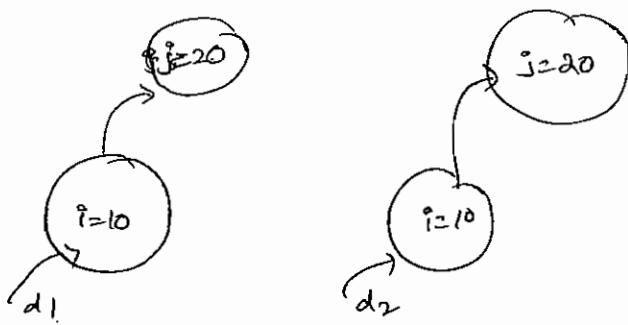
Shallow Cloning



vs Deep Cloning



Dog d2 = (Dog) d1.clone();

Deep cloning

`Dog d2 = (Dog)d1.clone();`

Shallow cloning

- The process of creating bitwise copy of an object is called **shallow cloning**.
- If the main object contains primitive variables then exactly duplicate copies will be created in the cloned object.
- If the main object contain any reference variable then correspond object won't be created just duplicate reference variable will be created pointing to old contained object.
- Object class `clone()` method meant for shallow cloning.

Program

```

class Cat {
    int j;
    Cat (int j) {
        this.j=j;
    }
}

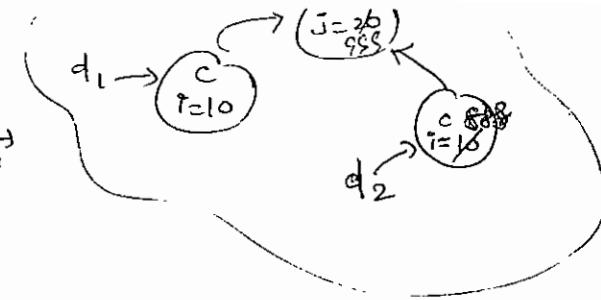
class Dog implements Cloneable {
    Cat c;
    int i;
    Dog (Cat c, int i) {
        this.c=c;
        this.i=i;
    }
}

public class Object {
    Object clone () throws CloneNotSupportedException {
        return super.clone();
    }
}

class ShallowCloning {
    public static void main (String [] args) throws CloneNotSupportedException {
        Cat c= new Cat(20);
    }
}
  
```

```

Dog d1 = new Dog(10);
d1.i = "----" + d1.c.j;
Dog d2 = (Dog)d1.clone();
d2.i = 888;
d2.j = 999;
System.out.println(d1.i);
}
    
```



→ In shallow cloning by using cloned object reference if we perform any change to the contained object then those changes will be reflected to the main object.

→ To overcome this problem we should go for deep cloning.

### Deep cloning

- The process of creating exactly duplicate independent copy including contained object is called deep cloning
- In Deep cloning if the main object contain any primitive variables then in the cloned object duplicate copies will be created
- If the main object contains any reference variable then the corresponding contained objects also will be created in the cloned copy.

By default Object class clone() method is meant for shallow cloning but we can implement deep cloning explicitly by overriding clone() method in our class.

### Program

```

class Cat {
    int j;
    Cat(int g) {
        this.j = g;
    }
}
    
```

```

class Dog implements Cloneable {
    Cat c;
    int i;
    Dog(Cat c, int i) {
        this.c = c;
        this.i = i;
    }
}
    
```

```

public Object clone() throws CloneNotSupportedException
{
    Cat c1 = new Cat("c-9");
    Dog d1 = new Dog(c1, 10);
    return d1;
}

class DeepCloning
{
    public static void main (String[] args) throws CloneNotSupportedException
    {
        Cat c = new Cat("c-20");
        Dog d1 = new Dog(c, 10);
        System.out.println("d1.i=" + d1.i);
        // Output - 20
        Dog d2 = (Dog)d1.clone();
        d2.i = 999;
        System.out.println("d2.i=" + d2.i);
        // Output - 999
    }
}

```

\* By using cloned object reference if we perform any change to the contained object then those changes won't be reflected to the main object.

Which cloning is Best?

If object contains only primitive variables then shallow cloning is the best choice  
If object contains reference variables then deep cloning is the best choice

### (5) getClass()

We can use getClass() method to get runtime class definition of an object.

**Public final Class getClass()**

By using this class-class object we can access class level properties like fully qualified name of the class, methods information, constructors information etc...

Program

Import java.lang.reflect.\*;

Class Test

{

    public void main (String[] args)

    {



        Object o = new String("dog");

```

Class c = o.getClass();
System.out.println("fully Qualified name of the class :" + c.getName());
method[] m = c.getDeclaredMethods();
System.out.println("methods information");
for(Method m1 : m)
{
    count++;
    System.out.println(m1.getName());
}
System.out.println("The number of methods is " + count);
}

```

Ex2: To display Database vendor specific Connection interface implemented class name?

```

Connection con = DriverManager.getConnection(...);
System.out.println(con.getClass().getName());

```

- Notes
- (1) After loading every .class file JVM will create an object of the type `java.lang.Class` in the heap area.
  - (2) We can use this class Object to get class level information  
→ `getClass()` method very frequently in reflections.
  - (3) finalize()

Just before destroying an object GarbageCollector calls `finalize()` method to perform cleanup activities.  
Once `finalize()` method completes automatically Garbage Collector destroys that object.

### Wait(), notify(), notifyAll()

We can use these methods for interthread communication.

The thread which is expecting updation, it is responsible to call `wait()` method. Then immediately the thread will enter into waiting state.

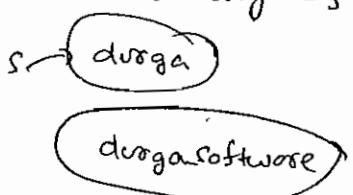
The thread which is responsible to perform updation, after performing updation the thread can call `notify()` method.  
The waiting thread will get that notification and continue its execution with those updates.

### ③ String (java.lang.String) class

#### Case 1

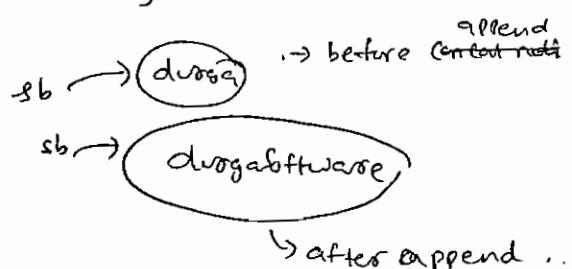
```
String s = new String("durga");
s.concat("software");
System.out.println(s); durga
s → durga
```

- Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created. This non-changeable behaviour is nothing but immutability of String.



```
StringBuffer sb = new StringBuffer("durga");
sb.append("software");
System.out.println(sb); durgasoftware
sb → durgasoftware
```

- Once we create StringBuffer object we can perform any change in the existing object this changeable behaviour is nothing but mutability of StringBuffer object.



#### Case 2

```
String s1 = new String("durga");
String s2 = new String("durga");
System.out.println(s1 == s2); false
System.out.println(s1.equals(s2)); true
s1 → durga
s2 → durga
```

- In String class .equals() method is overridden for content comparison hence even though objects are different if content is same .equals() method returns true

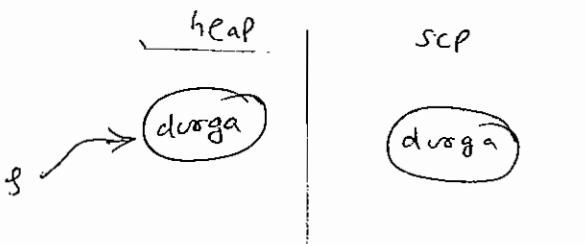
```
StringBuffer sb1 = new StringBuffer("durga");
StringBuffer sb2 = new StringBuffer("durga");
System.out.println(sb1 == sb2); false
System.out.println(sb1.equals(sb2)); true
sb1 → durga
sb2 → durga
```

- In StringBuffer class .equals() method is not overridden for content comparison hence if objects are different Object class .equals() method got executed which is used for reference comparison (Address comparison) due to this if object are different .equals() method returns false even though content is same.

### Case 3:

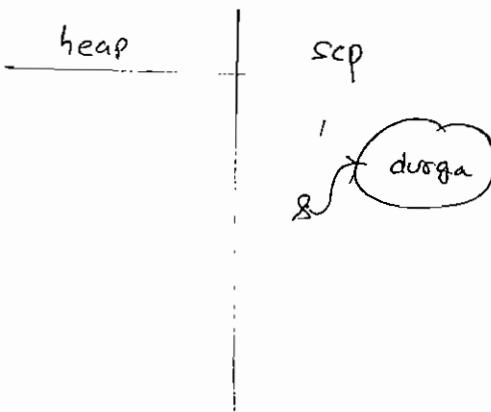
String s = new String ("durga")

In this case two objects will be created one in the heap area and the other in SCP (String Constant Pool) and s is always pointing to heap object.



String s = "durga";

In this case only one object will be created in SCP and s is always pointing to that object.



Note(1) Object creation in SCP is optional. First it will check if there any object already present in SCP with required content. If object already present then existing object will be reused.

If object not already available then only a new object will be created. But this rule is applicable only for SCP but not for the heap.

Note(2)

Garbage Collector is not allowed to access SCP area. Hence even though object doesn't contain reference variable it is not eligible for GC if it is present in SCP area.

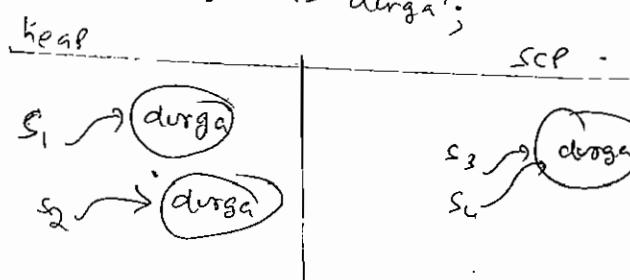
→ All SCP objects will be destroyed automatically at the time of JVM shutdown.

String s<sub>1</sub> = new String ("durga");

String s<sub>2</sub> = new String ("durga");

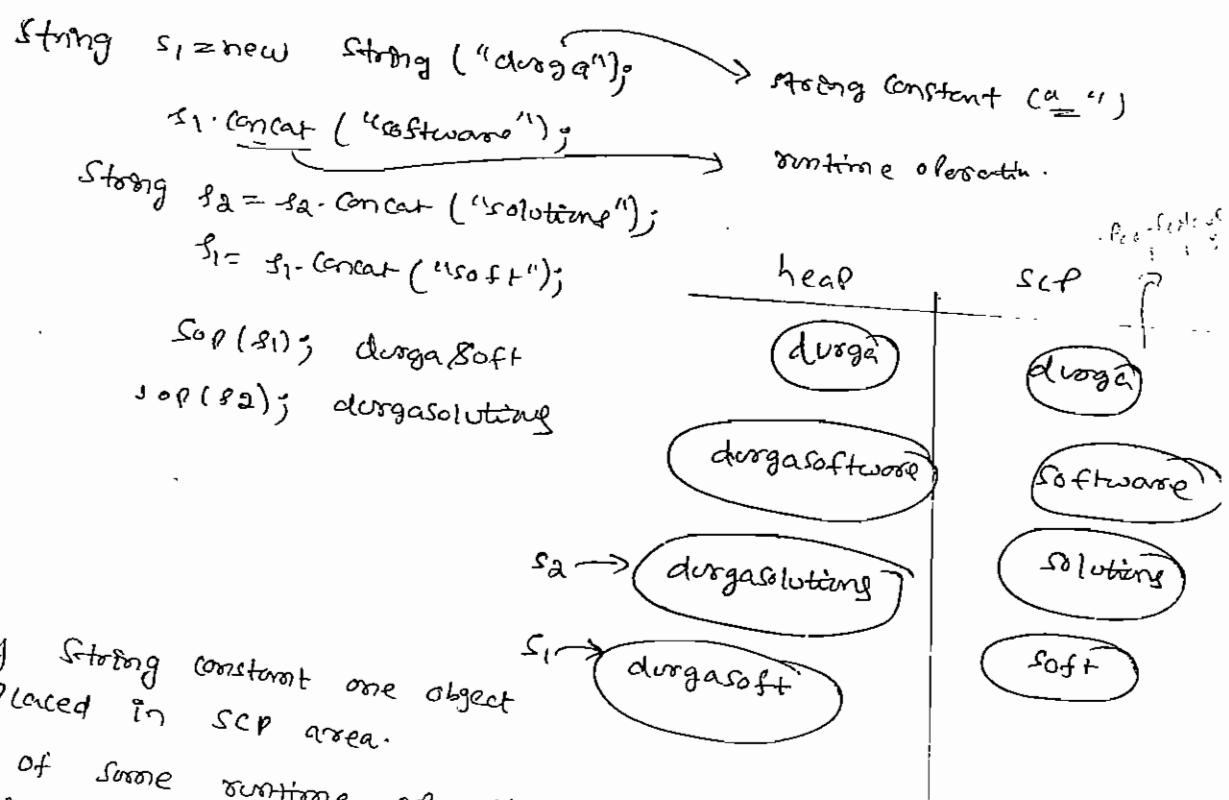
String s<sub>3</sub> = "durga";

String s<sub>4</sub> = "durga";



Note: Whenever we are using new operator compulsory a new object will be created in the heap area. Hence there may be a chance of existing two objects with same content in the heap area. but not in SCP. i.e. duplicate objects are possible in the heap area but not in SCP

Ex 3:



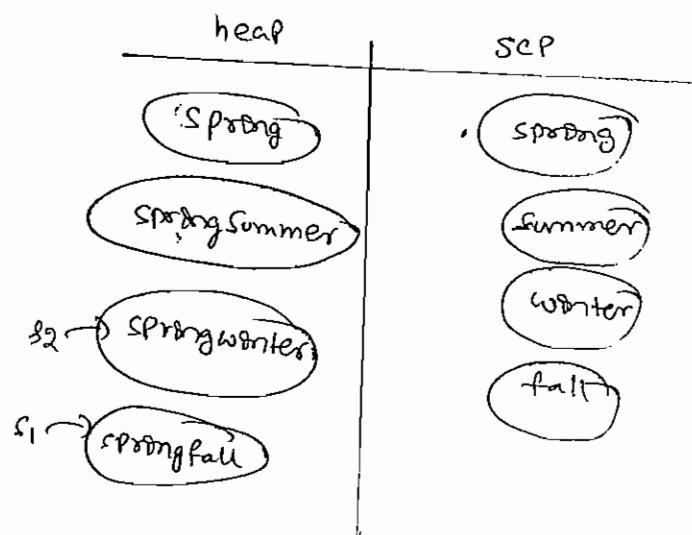
Note:

- (1) for every string constant one object will be placed in SCP area.
- (2) because of some runtime operation if an object is required to create that object in SCP area will be placed only in the heap area but not in SCP area.

Ex 4:

```

String s1 = new String("spring");
s1.concat("summer");
String s2 = s1.concat("winter");
s1 = s1.concat("fall");
s1 = s1.concat("face");
System.out.println(s1); // springfall
System.out.println(s2); // springwinter
    
```



Ex 5b

String  $s_1 = \text{new String("You can not change me!");}$

String  $s_2 = \text{new String(" You can not change me.");}$

$\text{sop}(s_1 == s_2); \text{ false}$

String  $s_3 = "You cannot change me.";$

$\text{sop}(s_1 == s_3); \text{ false}$

String  $s_4 = "You cannot change me.";$

$\text{sop}(s_2 == s_4); \text{ true}$

\* String  $s_5 = " \underline{\text{you can not}} + \underline{\text{change me}}"; \rightarrow ①$

$\text{sop}(s_2 == s_5); \text{ true}$

String  $s_6 = "you cannot";$

String  $s_7 = s_6 + "change me";$

{  
normal  
variable}

$\text{sop}(s_3 == s_7); \rightarrow \text{false}$

→ operation performed at compile time.  
 Compilation constant  
 Operation performed at runtime  
 to reduce the burden for JVM

$\text{sop}(2 + 3) \Rightarrow \text{sop}(5)$   
 $\text{sop}("ab" + "cd") \Rightarrow \text{sop}("abcd")$

{ normal variable  
may be changed  
future }

{  
operation performed  
at runtime }

{  
it is variable +  
change me constant  
operation performs  
at runtime }

final String  $s_6 = "You can not";$

String  $s_8 = s_6 + " \underline{\text{you cannot}}";$

$\text{sop}(s_3 == s_9);$  (3)

$\text{sop}(s_3 == s_9); \text{ true}$

$\text{sop}(s_6 == s_8); \text{ true}$

At line ①

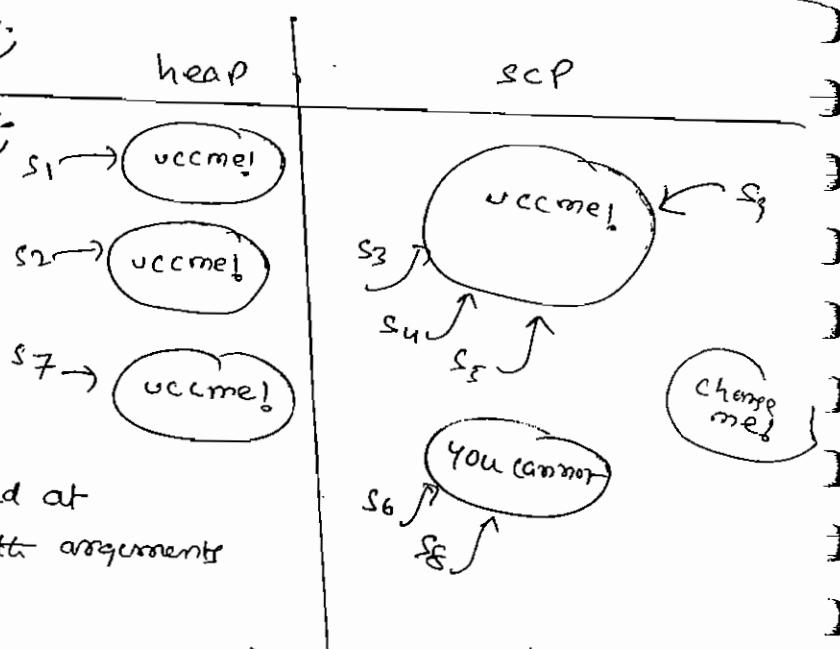
This operation will be performed at compiletime only because both arguments are compiletime constants

At line ②

This operation will be performed at Runtime only because atleast one argument is normal variable.

At line ③

This operation will be performed at ~~Runtime~~ compiletime only because both arguments are compiletime constants



## Interning of String Objects

We can use `intern()` method to get corresponding reference by using heap object reference `Scp Object`

(or)

By using heap object reference if we want to get corresponding `Scp Object` reference then we should go for `intern()` method.

Ex:

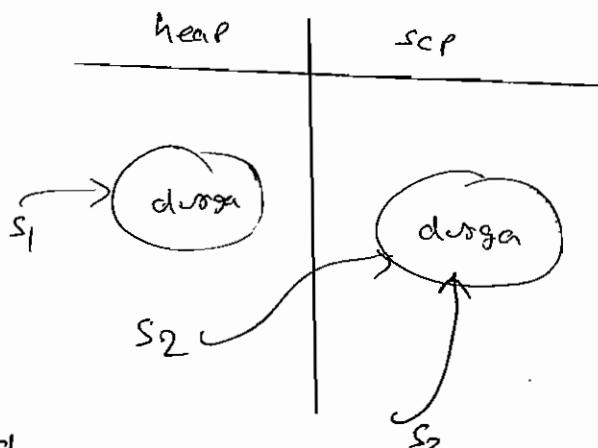
```
String s1 = new String("durga");
```

```
s1 = s1.intern();
```

```
Scp(s1 == s1); false
```

```
String s3 = "durga";
```

```
Scp(s2 == s3); true
```



② If the corresponding `Scp` object is not available then `intern()` method will create the corresponding `Scp` object.

Ex:

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

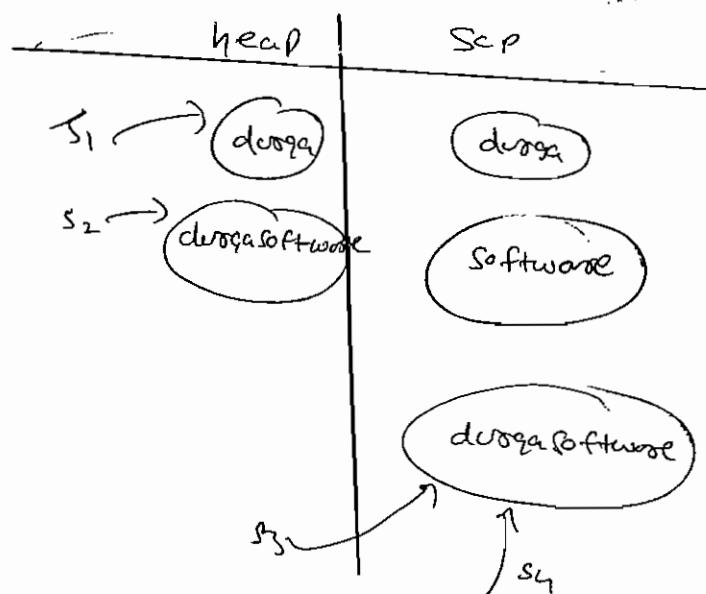
```
s1 = s1.concat("software");
```

```
String s3 = s1.intern();
```

```
Scp(s2 == s3); false
```

```
String s4 = "durgaSoftware";
```

```
Scp(s3 == s4); true
```



## Importance of String Constant Pool (SCP)

In our program if a string object is repeatedly required then it is not recommended to create separate object for every requirement because it creates performance and memory problems.

Instead of creating a separate object for every requirement we have to create only one object and we can reuse the same object for every requirement so that performance and memory utilization will be improved. This thing is possible because of SCP.

- (1) memory utilization of SCP are improved

But the major problem with SCP is, as several references pointing to the same object, by changing one reference if we are trying to change the content then remaining references will be affected to overcome this problem SUN people implemented Strong objects as immutable, i.e. once we creates existing object we can't perform any changes in the object if we are trying to perform any changes in the object a new object will be created. Hence SCP is the only reason for immutability of strong objects.

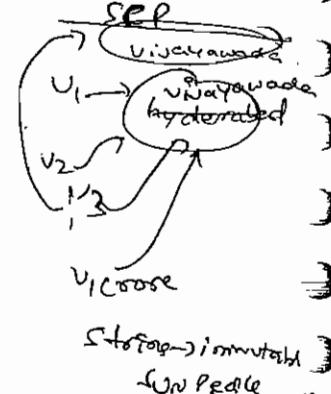
## FAQ's

- (1) what is the difference b/w String and StringBuffer?
- (2) Explain about Immutability and mutability with an example?
- (3) what is the difference b/w

String = new String ("durga"); and  
String = "durga";

Voter Registration form

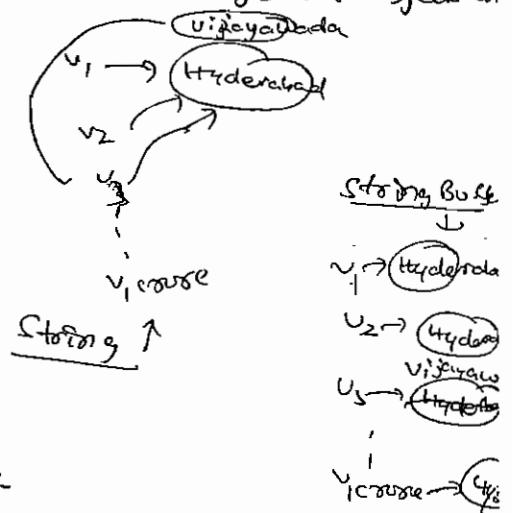
①	Name: Hiranyakshi	Photo
②	Father Name: Shyam Devji	
③	Date of Birth: 22-08-1986	
Age: 36		
Address:		
④	H.no: 1203 A-725	
⑤	Street: Banjara Hills	
⑥	Village/City: Hyderabad	
⑦	Mandal: -	
⑧	District: Rangareddy	
⑨	State: Telangana	
⑩	PIN: 5023002	
Identification marks:		
⑪	Identification mark 1: XYZ	
⑫	Identification mark 2: ABC	
Submit		



- (4) Other than immutability and mutability is any other difference b/w String and StringBuffer? (equivalent)
- (5) What is SCP?
- It is a specially designed memory area for String objects.
- (6) What is the advantage of SCP?
- (7) What is the disadvantage of SCP?
- SCP is the only reason for the immutability of String object.
- (8) Why SCP like concept is available only for String but not for StringBuffer?
- String is the most commonly used object ~~other~~ and hence Sun people provided special memory management for String object but StringBuffer is not commonly used object and hence special memory management is not required for StringBuffer.
- (9) Why String Objects are immutable?

whereas StringBuffer objects are

In the case of String because of SCP a single object can be referenced by multiple references. By using one reference if we are allowed to change the content in the existing object then remaining references will be affected to overcome this problem Sun people implement String objects as immutable.



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 changes with those changes a new object will be created. But in StringBuffer there is no concept like SCP hence for every requirement a separate object will be created by using one reference if we are trying to change the content then there is no effect on remaining references hence immutability concept not required for the StringBuffer.

In addition to String objects any other objects are immutable in Java. In addition to String objects all wrapper class objects also are immutable.

Q(1) Is it possible to create our own immutable class?

Yes

Q(2) How to create our own immutable class? Explain with an example?

Q(3) Immutable means non-changable where as final means also non-changable then what is the difference b/w final and immutable?

## Constructors of String class

API

① `String s = new String();`

Creates an empty String object

`String s = new String(String literal);`

Creates a String object on the heap for the given String literal

③ `String s = new String(StringBuffer sb);`

Creates an equivalent String object for the given StringBuffer

④ `String s = new String(char[] ch);`

Creates an equivalent String object for the given  
char Array

Ex:- `char[] ch = {'a', 'b', 'c', 'd'};`

`String s = new String(ch);`

`sop(s); abcde`

⑤ `String s = new String(byte[] b);`

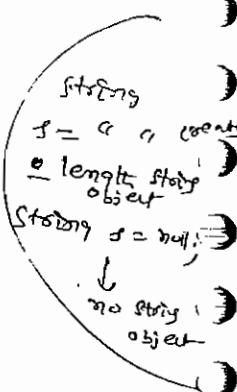
Creates an equivalent String object for the given byte array

Ex:- `byte[] b = {100, 101, 102, 103};`

`String s = new String(b);`

`sop(s); defg`

d - 100  
e - 101  
f - 102  
g - 103



Important methods of String class

① **Public char charAt(int index)**

returns the character located at specified index

Ex: String s = "durga";

Sop(s.charAt(3)); g

Sop(s.charAt(30));

RE: **StringIndexOutOfBoundsException**

② **Public String concat(String s)**

The overloaded + and += operators also meant for concatenation purpose only.

Ex: String s = "durga";

s = s.concat(" software");

// s = s + " software";

// s += " software";

Sop(s); // durgasoftware

③

**Public boolean equals(Object o)**

To perform content comparison where case is important

This is overriding version of Object class equals() method.

④ **Public boolean equalsIgnoreCase(String s)**

To perform content comparison where case is not important

Ex:

String s = "java";

Sop(s.equals(" JAVA")); false

Sop(s.equalsIgnoreCase("JAVA")); true.

Note

In general we can use equalsIgnoreCase method to validate user name where case is not important.

Where as we can use equals() method to validate password where case is important.

⑤ public String substring (int begin)

returns substring from begin index to end of the string

⑥ public String substring (int begin, int end);

returns substring from begin index to end-1 index

Eg:  
String s = "abcdefg";

Sop (s.substring (3)); // defg

Sop (s.substring (2, 5)); // cdef

⑦ public int length()

returns number of characters present in the string

Eg:  
String s = "charge";

Sop (s.length())

Sop (s.length()); // 5

C++ can not find symbol  
symbol : variable length  
location : g::length

Note

length variable applicable for arrays but not for string objects  
where as length() method applicable for string objects but not  
for arrays

⑧ public String replace (char oldch, char newch)

Eg:  
String s = "ababi";

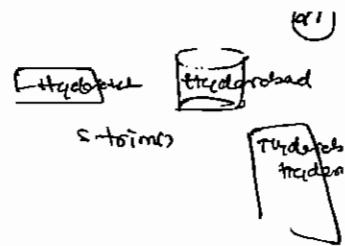
:: Sop (s.replace ('a', 'b')) ; // bbbbb

⑨ public String toLowerCase();

⑩ public String toUpperCase();

(11) Public String trim();

To remove blank spaces present at beginning and end of the string but not middle blank spaces



(12) Public int indexOf(char ch);

Returns index of first occurrence of specified character

(13) Public int lastIndexOf(char ch);

String s = "ababa";

System.out.println(s.indexOf('a')) ; //0

System.out.println(s.lastIndexOf('a')) ; //4

X

### Note

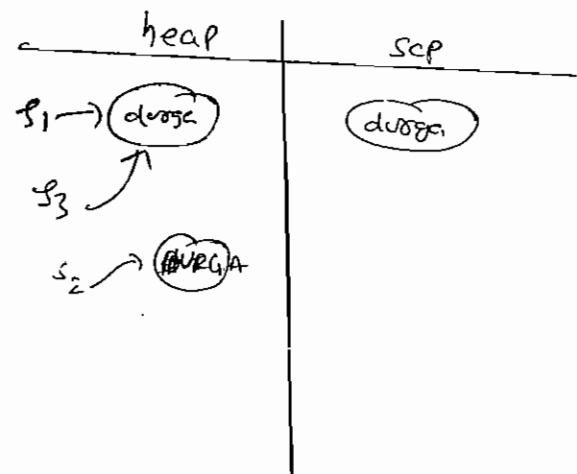
Because of Runtime operation if there is a change in the content then with those changes a new object will be created on the heap.

If there is no change in the content then existing object will be reused and new object won't be created

→ Whether the object present in heap or SCP the rule is same

Ex:

```
String s1 = new String("George");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
System.out.println(s1 == s2); false
System.out.println(s1 == s3); true
```

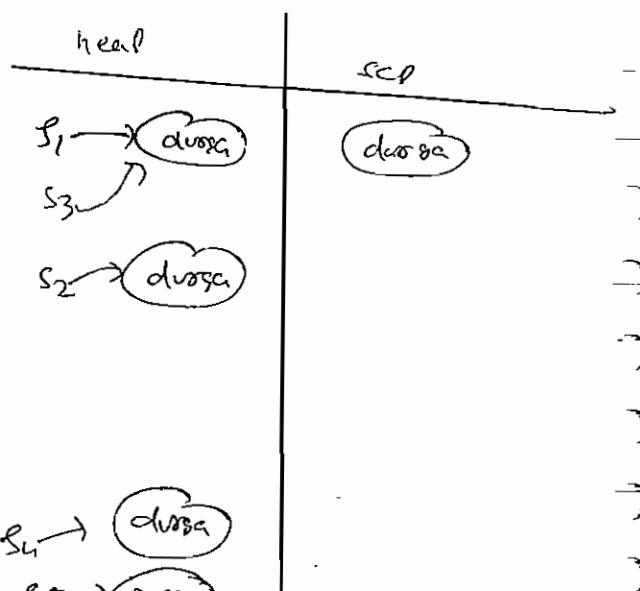


Ex2

```

String s1 = new String("durga");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
{
    String s4 = s2.toLowerCase();
    String s5 = s4.toUpperCase();
}

```



Ex3:

```

String s1 = "durga";
String s2 = s1.toString();
System.out.println(s1 == s2); // true

```

```

String s3 = s1.toLowerCase();

```

```

String s4 = s1.toUpperCase();

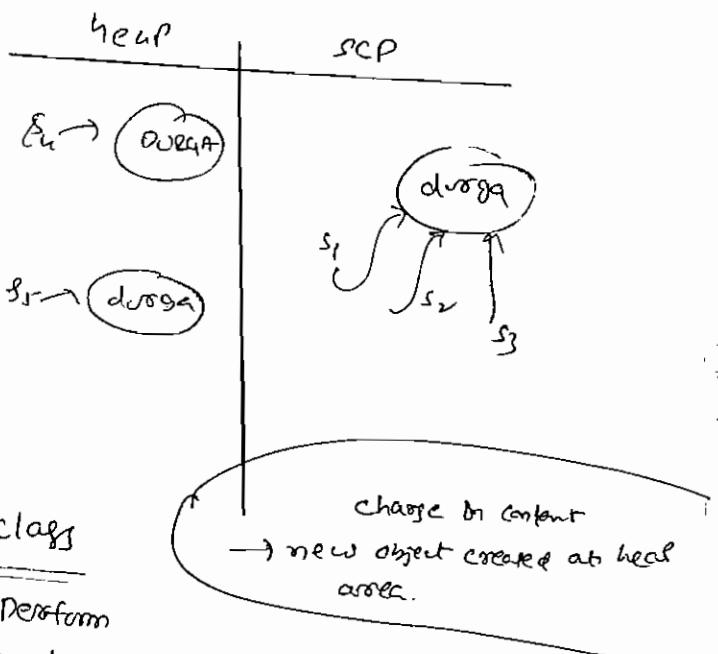
```

```

String s5 = s4.toLowerCase();

```

String  
no change in content



### How to create our own immutable class

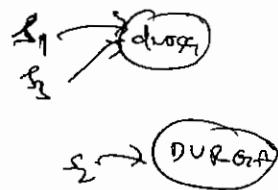
Once we creates an object we can't perform any changes in that object. If we are trying to perform any change and if there is a change in the content then with those changes a new object will be created. If there is no change in the content then existing object will be reused. This behaviour is nothing but immutability.

Ex:

```

String s1 = new String("durga");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();

```





Hence final and immutable both are different concepts.

Ex

```
final StringBuffer sb = new StringBuffer("class");  
    ^ reference variable
```

```
sb.append("software");
```

```
for (sb); // change software
```

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

sb →

class software

e.g.: Can not assign a value to  
final variable sb

which of the following are meaningful

- ✓ ① final variable
- ✗ ② immutable variable
- ✗ ③ final object
- ✓ ④ immutable object

08/09/14

## StringBuffer

then it is recommended to go for String.

- if the content is not fixed and keep on changing then it is not recommended to use String. because for every change a new object will be created which effects performance of the system.
- to handle this requirement we should go for StringBuffer.
- The main advantage of StringBuffer over String is all required changes will be performed in the existing object only.

## Constructors

① StringBuffer sb = new StringBuffer();

Creates an empty StringBuffer object with default initial capacity 16 once be StringBuffer reaches its max capacity a new StringBuffer object will be created with  $\boxed{\text{new capacity} = (\text{current capacity} + 1) \times 2}$

Ex:  
StringBuffer sb = new StringBuffer();  
System.out.println(sb.capacity()); // 16  
sb.append("abcdefghijklmnopqrstuvwxyz");  
System.out.println(sb.capacity()); // 16  
sb.append("q");  
System.out.println(sb.capacity()); // 34

② StringBuffer sb = new StringBuffer(*int initialCapacity*);

Creates an empty StringBuffer object with specified initial capacity

③ StringBuffer sb = new StringBuffer(*String s*);  
Creates an equivalent StringBuffer for the given String with  
 $\boxed{\text{capacity} = \text{s.length}() + 16}$

Ex: StringBuffer sb = new StringBuffer("durga");  
System.out.println(sb.capacity()); // 21

≤ 5 + 16

## Important methods of StringBuffers

- ① public int length();
- ② public int capacity();
- ③ public char charAt(int index);

Exe

```
StringBuffer sb = new StringBuffer("durga");
Sop(sb.charAt(3)); // g
Sop(sb.charAt(30));
RE: StringIndexOutOfBoundsException
```

- ④ public void setCharAt (int index, char ch)

To replace the character located at specified index with provided character.

- ⑤ public StringBuffer append (String s)

(int i)  
(long l)  
(char ch)  
(boolean b)

} overlaoded Methods

Ex: StringBuffer sb = new StringBuffer();  
 sb.append("PI value is : ");  
 sb.append(3.14);  
 sb.append(" It is exactly: ");  
 sb.append(true);  
 Sop(sb);

t added at last

⑥

- public StringBuffer insert (int index, String s)

(int index, int i)  
(int index, double d)  
(int index, char ch)  
(int index, boolean b)

} overlaoded methods

Ex: StringBuffer sb = new StringBuffer("abcdefg");  
 sb.insert(2, "xyz");  
 Sop(sb); // abcxyzdefg

- (7) public StringBuffer delete (int begin, int end)  
To delete characters located from begin index to end-1 index.
- (8) public StringBuffer deleteCharAt (int index)  
To delete the character located at specified index.
- (9) public StringBuffer reverse ();  
Ex: StringBuffer sb = new StringBuffer ("durga");  
sb.reverse(); agnud
- (10) public void setLength (int length);  
Ex: StringBuffer sb = new StringBuffer ("aishwaryaabhi");  
sb.setLength (8);  
SOP (sb); // aishwarya.
- (11) public void ensureCapacity (int capacity);  
to increase capacity on fly based on our requirement  
Ex: StringBuffer sb = new StringBuffer();  
SOP (sb.capacity()); // 16  
sb.ensureCapacity (100);  
SOP (sb.capacity()); // 100
- (12) public void trimToSize();  
to deallocate extra allocated free memory  
Ex: StringBuffer sb = new StringBuffer(100);  
sb.append ("abc");  
sb.trimToSize();  
SOP (sb.capacity()); // 3



### StringBuilder

Every method present in StringBuffer is synchronized and hence only one thread is allowed to operate on StringBuffer object at a time. which may creates performance problems to handle this requirement Sun people introduced StringBuilder concept in 1.5 version

`StringBuilder` is exactly same as `StringBuffer` except the following differences.

<code>StringBuffer</code>	<code>StringBuilder</code>
① Every method present in <code>StringBuffer</code> is synchronized	① Every method present in <code>StringBuilder</code> is non synchronized
② At a time only one thread is allowed to operate on <code>StringBuffer</code> object and hence <code>StringBuffer</code> object is thread safe.	② At a time multiple threads are allowed to operate on <code>StringBuilder</code> object and hence <code>StringBuilder</code> is not thread safe.
③ Threads are required to wait to operate on <code>StringBuffer</code> object and hence relatively performance is low	③ Threads are not required to wait to operate on <code>StringBuilder</code> object and hence relatively performance is high
④ <code>StringBuffer</code> class introduced in 1.0 version	④ introduced in 1.5 version

Note: Except the above differences everything is same in `StringBuffer` and `StringBuilder` (excluding methods and constructors)

### `String` vs `StringBuffer` vs `StringBuilder`

- ① If the content is fixed and won't change frequently then we should go for `String`
- ② If the content is not fixed and keep on changing but thread safety required then we should go for `StringBuffer`.
- ③ If the content is not fixed keep on changing but thread safety is not required then we should go for `StringBuilder`.

### Method Chaining

For most of the methods in `String`, `StringBuffer` and `StringBuilder` return types are same type. Hence after applying a method on the result we can call another method which forms method chaining.

`sb.m1().m2().m3().m4().....`

In method chaining method calls will be executed from left to right

```
StringBuffer sb = new StringBuffer();
sb.append("durga").append("software").append("solutions").insert(2, "xyz");
sb.reverse().delete(2, 10);
System.out.println(sb);
```

## Wrapper classes

The main objectives of Wrappers classes are

- (1) To wrap primitive into object form so that we can handle primitives also just like objects.
- (2) To define several utility methods which are required for primitives

### Constructors

Almost all wrapper classes corresponding primitive as argument. Containing (2) constructors one can take String as argument and the other can take

#### Ex: (1)

Integer I = new Integer(10);  $\rightarrow$  int primitive

Integer I = new Integer("10");  $\rightarrow$  String

#### (2)

Double D = new Double(10.5);

Double D = new Double("10.5");

\* If the String argument not representing a number then we will get RuntimeException saying NumberFormatException

Ex: Integer I = new Integer("ten");

Result: NumberFormatException

\* Float class containing (3) constructors with float, double & string arguments

Ex: float f = new float(10.5f);

float f = new float("10.5f");

float f = new float("10.5");

float f = new float("10.5");

\* Character class containing only one constructor which can take char argument

Ex: character ch = new Character('a');

Character ch = new Character('a');

(1)  $\left\{ \begin{array}{l} \text{Integer I = new Integer();} \\ \text{ArrayList L = new ArrayList();} \\ \text{L.add(10); X} \\ \text{L.add(2)} \end{array} \right.$

(2) class ArrayList  
    {  
        set()...  
        get()...  
    }  
    class Integer  
    {  
        int value;  
        String str;  
        Integer add(Integer);  
        Integer subtract(Integer);  
        Integer multiply(Integer);  
        Integer divide(Integer);  
    }

String s = "10";  $\rightarrow$  Integer

Boolean class contains ② Constructors one can take primitive as argument and the other can take String argument.

→ If we pass Boolean primitive as argument the only allowed values are true (or) false where case is important and content is also important.

Ex: 1) Boolean B = new Boolean(true); ✓

Boolean B = new Boolean(false); ✓

✗ Boolean B = new Boolean (True); ]

✗ Boolean B = new Boolean (durga); ]

→ if we are passing String type as argument then case and content both are not important

if the content is case insensitive String of "true" then it is treated as true otherwise it is treated as false.

Ex:

Boolean B = new Boolean ("true"); true

Boolean B = new Boolean (" True "); true

Boolean B = new Boolean (" TRUE "); true

Boolean B = new Boolean (" MalikA "); false

Boolean B = new Boolean (" malika "); false

Boolean B = new Boolean (" Jaseena "); false

Boolean B = new Boolean (" jaseena "); false

Ex: (Lab Test)

↳ p s v main (String[] args)

{ Boolean x = new Boolean (" Yes ");

Boolean y = new Boolean (" No ");

System.out(x); false

System.out(y); false

System.out(x.equals(y)); // true

wrapper class

Corresponding constructor Arguments

Byte → byte or String

Short → short or String

Integer → int or String

Long → long or String

\* Float → float or String as double

Double → double or String

\* Character → char or String

\* Boolean → boolean or String

- Note → In all wrapper classes `toString()` method is overridden to return content directly.
- In all wrapper classes `equals()` method is overridden for content comparison.

## Utility Methods

- ① `valueOf()`
- ② `xxxValue()`
- ③ `parseXXX()`
- ④ `toString()`

### ① valueOf()

we can use `valueOf()` methods to create wrapper object for the given primitive (or) string

format

Every wrapper class except character class contains a static valueOf() method to create wrapper object for the given string

Public static wrapper valueOf(String s)

Ex:

Integer I = Integer.valueOf("10");  
 Double D = Double.valueOf("10.5");  
 Boolean B = Boolean.valueOf("true");

format:

Every Integral type wrapper class (Byte, Short, Integer, Long) contains the following valueOf() method to create wrapper object for the given specified radix string.

Specified  
radix  
base

Public static wrapper valueOf(String s, int radix);

The allowed range of radix is 2 to 36

① Integer I = Integer.valueOf("100", 2);  
 System.out.println(I); // 4

② Integer I = Integer.valueOf("101", 4);  
 System.out.println(I); // 17

$$\begin{array}{r} \text{101}_4 \\ \times 4^0 \\ \hline \text{101}_4 \\ \end{array}$$

$$\begin{array}{r} \text{101}_4 \\ \times 4^1 \\ \hline \text{101}_4 \\ - 16 \\ \hline \text{17}_4 \end{array}$$

base 2: 0, 1  
 base 3: 0 to 2  
 base 8: 0 to 7  
 base 10: 0 to 9  
 base 11: 0 to A  
 base 16: 0 to F, 0  
 base 36: 0 to 9, A to Z

Form 3.5. Every wrapper class including character class contains

a static `valueOf()` method to create wrapper object for the given primitive

```
public static wrapper valueOf(primitive P);
```

Exs

```
Integer I = Integer.valueOf(10);
```

```
Character ch = Character.valueOf('a');
```

```
Boolean B = Boolean.valueOf(true);
```



(2)

xxxValue()

We can use `xxxValue()` methods to get primitive for the given wrapper object.

Every number type wrapper class (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`) containing the following ⑥ methods to get primitive for the given wrapper object.

- ① public byte `byteValue()`
- ② public short `shortValue()`
- ③ public int `intValue()`
- ④ public long `longValue()`
- ⑤ public float `floatValue()`
- ⑥ public double `doubleValue()`

Exs

```
Integer I = new Integer(130);
```

```
System.out.println(I.byteValue()); -128
```

```
System.out.println(I.shortValue()); 130
```

```
System.out.println(I.intValue()); 130
```

```
System.out.println(I.longValue()); 1300000000000000000L
```

```
System.out.println(I.floatValue()); 130.0
```

```
System.out.println(I.doubleValue()); 130.0
```

Integer  
Byte  
Character  
Long  
Float  
Double  
byteValue()  
shortValue()  
intValue()  
longValue()  
floatValue()  
doubleValue()

charValue()  
Character class contains charValue() method to get char primitive for the given Character object.

Public char charValue()

Ex:

```
Character ch = new Character('a');
char c = ch.charValue();
System.out.println(c);
```

booleanValue()

Boolean class containing booleanValue() method to get boolean primitive for the given Boolean object.

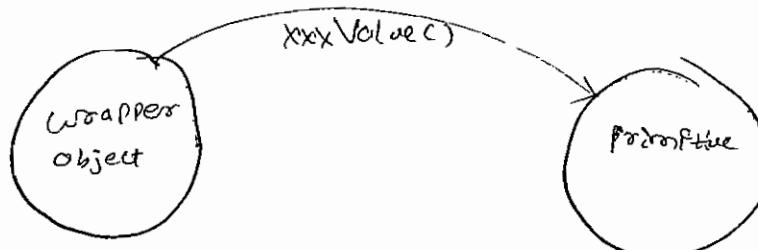
Public boolean booleanValue();

Ex:

```
Boolean b = Boolean.valueOf("false");
boolean b1 = b.booleanValue();
System.out.println(b1);
```

Note

In total  $38 (= 6 \times 6 + 1 + 1)$  XXXValue() methods are possible.



evening 5PM

08/09/14

(3) parseXXX()

We can use parseXXX() methods to convert String to primitive

Form - I  
Every wrapper class except Character class contains the following ParseXXX() method to find primitive for the given String object

Public static Primitive parseXXX(String s);

Ex:

```
int i = Integer.parseInt("10");
double d = Double.parseDouble("10.5");
boolean b = Boolean.parseBoolean("true");
```

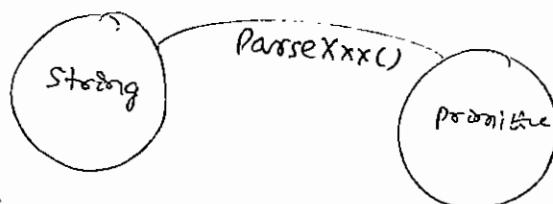
Form 2 Every Interceptor wrapper class (Byte, short, Integer, Long) contains the following ParseXXX() method to convert specified radix string to primitive.

Public static Primitive parseXXX(String s, int radix);

The allowed range of radix is 2 to 36

e.g.

```
int i = Integer.parseInt("1111", 2);  
System.out.println(i);
```



7)

toString()

We can use toString() method to convert wrapper object (or) primitive to string

Form - I

Every wrapper class containing the following toString() method to convert wrapper object to String type.

Public String toString()

It is the overriding version of Object class toString() method whenever we are trying to print wrapper object reference internally this toString() method will be called.

```
Integer I = new Integer(10);  
String s = I.toString();  
System.out.println(s); // 10  
System.out.println(I); // 10
```

Form - II

Every wrapper class including character class containing the following static toString() method to convert primitive to String

Public static String toString(Primitive p)

```
String s = Integer.toString(10);  
String s = Boolean.toString(true);  
String s = Character.toString('a');
```

form-3 Integer and Long classes contains the following toString() method to convert primitive to specified radix string.

Public static String toString(Primitive p, int radix);

The allowed range of radix : 2 to 36

String s = Integer.toString(15, 2);  
sop(s); // 1111

form-4

toXXXString()

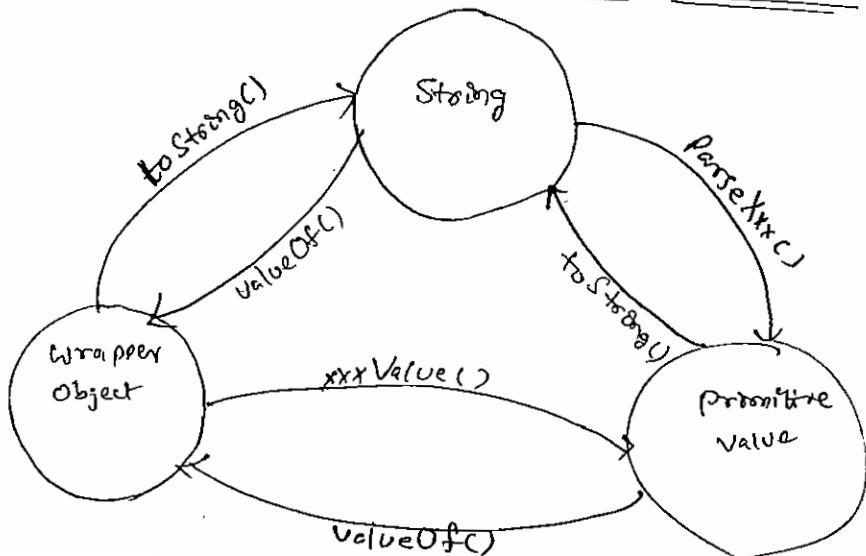
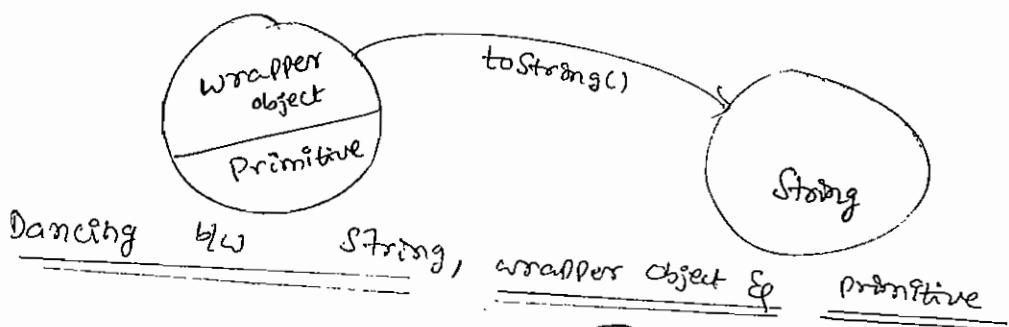
Integer & Long classes contains the following toXXXString() methods

- ① Public static String BinaryString(primitive p)
- ② Public static String toOctalString(Primitive p)
- ③ Public static String toHexString(Primitive p)

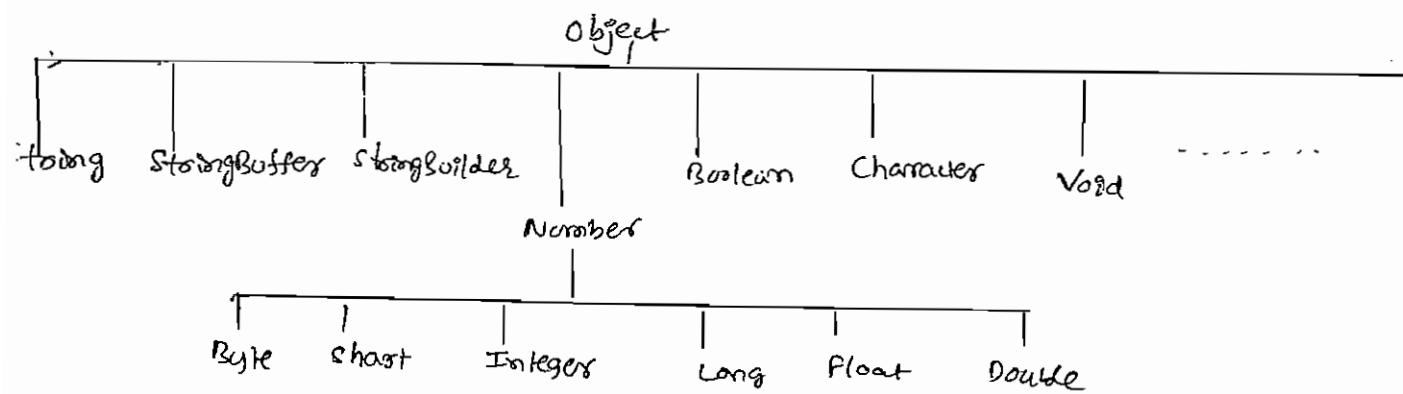
String s = Integer.BinairyString(10);  
sop(s); // 1010

String s = Integer.toOctalString(10);  
sop(s); // 12

String s = Integer.toHexString(10);  
sop(s); // a



## Partial hierarchy of java.lang package



### Conclusion

- ① The wrapper classes which are not child class of Number ~~(e.g.)~~ are Boolean & Character.
- ② The wrapper classes which are not direct child class of Object are Byte, Short, Integer, Long, Float, Double.
- ③ String, StringBuffer, StringBuilder and all wrapper classes are final classes.
- ④ In addition to String objects all wrapper class objects also immutable.
- ⑤ Some times Void class is also considered as wrapper class.

### Void class

- It is a final class and it is the direct child class of Object.
- It doesn't contain any methods and it contains only one variable.  
Void.TYPE
- In General we can use Void class in reflections to check whether the method return type is Void or not.
- Ex: if (getmethod("m1").getReturnType() == Void.TYPE)  

```

if (getmethod("m1").getReturnType() == Void.TYPE)
{
    ...
}
  
```
- Void is the class representation of void keyword in java.

## AutoBoxing & AutoUnBoxing

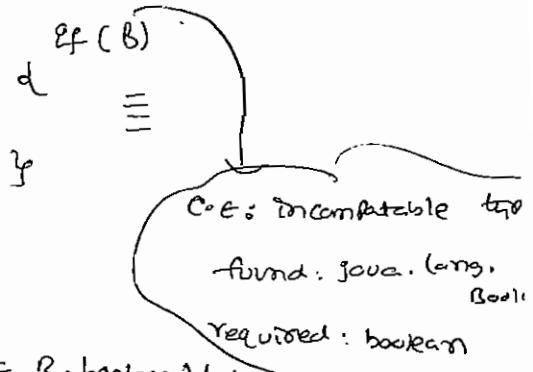
until 1.4 version we can't provide primitive in the place of wrapper object and wrapper object in the place of primitive all required conversions should be performed explicitly by the programmer

Ex:

```
ArrayList l = new ArrayList();
l.add(10); → C.E
```

```
Integer I = new Integer(10);
l.add(I);
```

```
Boolean B = new Boolean(true);
```



\* But from 1.5 version onwards we can provide primitive value in the place of wrapper object and wrapper object in the place of primitive all required conversions will be performed automatically by the compiler these automatic conversions are nothing but autoboxing and autounboxing.

### AutoBoxing

Automatic conversion of primitive to wrapper object by compiler is called AutoBoxing

e.g.: `Integer I = 10;` (Compiler converts int to Integer automatically by AutoBoxing)

After compilation the above line will become

`Integer I = Integer.valueOf(10);` i.e. internal auto boxing concept implemented by using valueOf() method

### AutoUnBoxing

Automatic conversion of wrapper object to primitive by compiler is called Auto unboxing

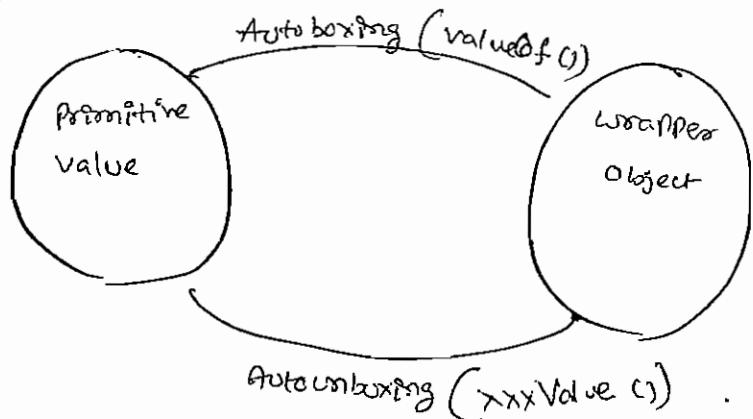
```
Integer I = new Integer(10);
```

`int q = I;` (Compiler converts Integer to int automatically by Auto unboxing)

after compilation the above line will become

`int q = I.intValue();`

i.e. internally Autounboxing concept is implemented by using `xxxValue()` methods.



Ex 1

```
class Test
{
    static Integer I=10; → ① Autoboxing
    public void main(String[] args)
    {
        int i = I; → ② Autounboxing
        m(i);
    }
    public void m(Integer k) → ③ Autounboxing
    {
        int m=k; → ④ Autounboxing
        System.out.println(m);
    }
}
```

(Java - syntax review)

At 8<sup>th</sup> valid in 1.5 version  
but invalid in 1.4 version

Note 8

Just because of Autoboxing and Autounboxing we can use primitives and wrapper objects interchangeably from 1.5 version onwards.

Ex 2

```
class Test
{
    static Integer I=@;
    public void main(String[] args)
    {
        int m=I;
        System.out.println(m);
    }
}
```

class Test
{
 static Integer I; → (I → object default value = null)
 public void main(String[] args)
 {
 int m=I; → (I → object default value = null)
 System.out.println(m);
 }
}

RF: NPE

int m=I.intValue();  
(null)

Note 9

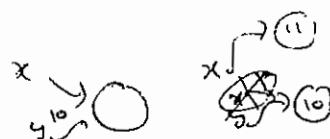
On null reference if we are trying to perform Autounboxing then we will get Runtime exception saying `NullPointerException`

Ex 3

```

Integer x=10;
Integer y=x;
x++;
System.out.println(x);
System.out.println(y);
System.out.println(x==y); false

```



∴ all wrapper class objects are immutable

Notes

All wrapper class objects are immutable i.e. once we created wrapper class object we can't perform any changes in that object if we are trying to perform any changes with those changes a new object will be created

Ex 8

① Integer x = new Integer(10); x → (10)  
 Integer y = new Integer(10); y → (10)  
 System.out.println(x==y); false ✓

② Integer x = new Integer(10);  
 Integer y = 10; x → (10)  
 System.out.println(x==y); y → (10) ✓

∴ new Integer  
↳ new object created

③ Autoboxing ↳  
Autoboxing already created autoboxing

④ Integer x = 10;  
 Integer y = 10; x → (10)  
 System.out.println(x==y); y → (10) ✓

⑤ Integer x = 100;  
 Integer y = 100; x → (100)  
 System.out.println(x==y); true ✓

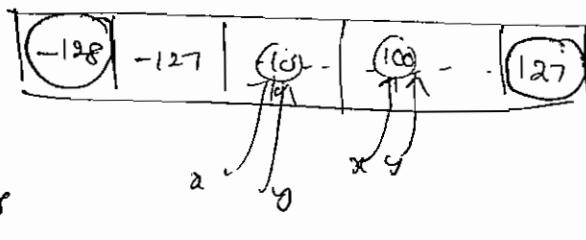
⑥ Integer x = 1000;  
 Integer y = 10000; x → (1000)  
 System.out.println(x==y); y → (1000) ✓

Conclusion:

Internally to provide support for autoboxing a buffer of wrapper objects will be created at the time of wrapper class loading. by Autoboxing if an object is required to create first JVM will check whether this object already present in the buffer or not if it is already present in the buffer then existing buffer object will be used. if it is not already available in the buffer then JVM will create a new object.

class Integer

{  
  static  
  2



- But Buffer concept is available only on the following ranges

Byte	→ always
Short	→ -128 to 127
Integer	→ -128 to 127
Long	→ -128 to 127
Character	→ 0 to 127
Boolean	→ always

Float  
Double  
all non-integer values is there

- Except this range in all remaining cases a new object will be created

Ex:

① Integer x = 127;  
  Integer y = 127;  
  System.out.println(x == y); true

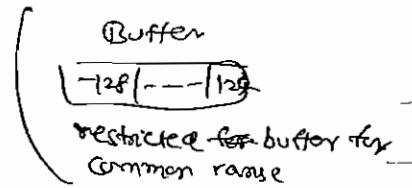
② Integer x = 128;  
  Integer y = 128;  
  System.out.println(x == y); false

③ Boolean x = false;  
  Boolean y = false;  
  System.out.println(x == y); true

Double x = 10.0;  
Double y = 10.0;  
System.out.println(x == y); false

X

\* Internally hence buffering concept is implemented by using valueOf methods  
autoboxing concept is applicable for valueOf() method also



class Integer  
{  
  static  
  1 byte / to + comp  
  // Float / Double  
  3  
  Create 2 objects  
  already waste  
  so mostly commonly  
  used some Buffer  
  concept is the  
  -128 to 127

```

Integer x = new Integer(10);
Integer y = new Integer(10);
System.out.println(x == y); // false
new operator creates new object
    
```

```

Integer x = 10;
Integer y = 10;
System.out.println(x == y); // true
        
```

```

Integer x = Integer.valueOf(10);
Integer y = Integer.valueOf(10);
System.out.println(x == y); // true
    
```

AutoBoxing

```

Integer x = 10; // internally concrete
Integer y = Integer.valueOf(10);
System.out.println(x == y); // true
    
```

## Overloading with respect to

## AutoBoxing, widening & narrowing methods

### Case 1:

#### AutoBoxing vs Widening

##### class Test

```

public static void m1(Integer i)
{
    System.out.println("AutoBoxing");
}

public static void m1(long l)
{
    System.out.println("widening");
}
    
```

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

```

int x = 10;
    
```

```

m1(x); // O/P: widening
    
```

widening dominates AutoBoxing

### Case 2:

#### Widening vs var-arg method

##### class Test

```

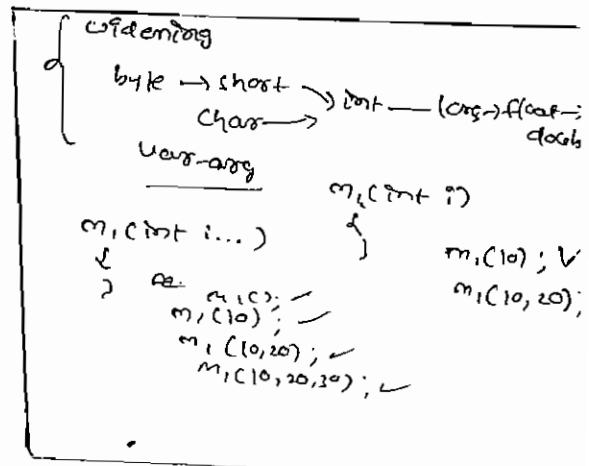
public static void m1(int... x)
{
    System.out.println("var-arg method");
}

public static void m1(long l)
{
    System.out.println("widening");
}
    
```

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

```

int x = 10; // O/P: widening
    
```



A. Boxing

B. Widening

1995

2014

19+ year

old concept  
widening dominates var-arg methods

O/P: widening

## Case 3: Autoboxing vs var-arg

class Test

```

d public static void m1(int... x)
{
    System.out.println("Var-arg method");
}

public static void m1(Integer i)
{
    System.out.println("Autoboxing");
}

public static void main(String[] args)
{
    int x=10;
    m1(x);
}

```

(P: Autoboxing)

Autoboxing dominates var-arg methods

→ In General var-arg method will get least priority i.e if no other method matched then only var-arg method will get the chance. It is exactly same as default case inside switch.

### Note

While resolving overloaded methods compiler will always give the precedence in the following order

- (1) Widening
- (2) Autoboxing
- (3) var-arg methods.

## Case 4:

class Test

```

d public static void m1(Long l)
{
    System.out.println("Long");
}

```

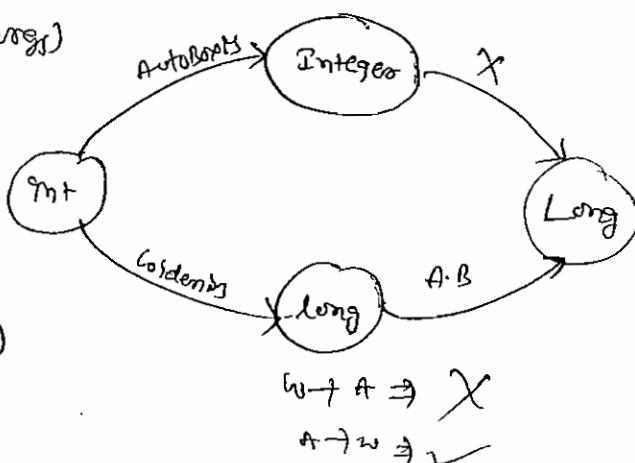
```

public static void main(String[] args)
{
    int x=10;
    m1(x);
}

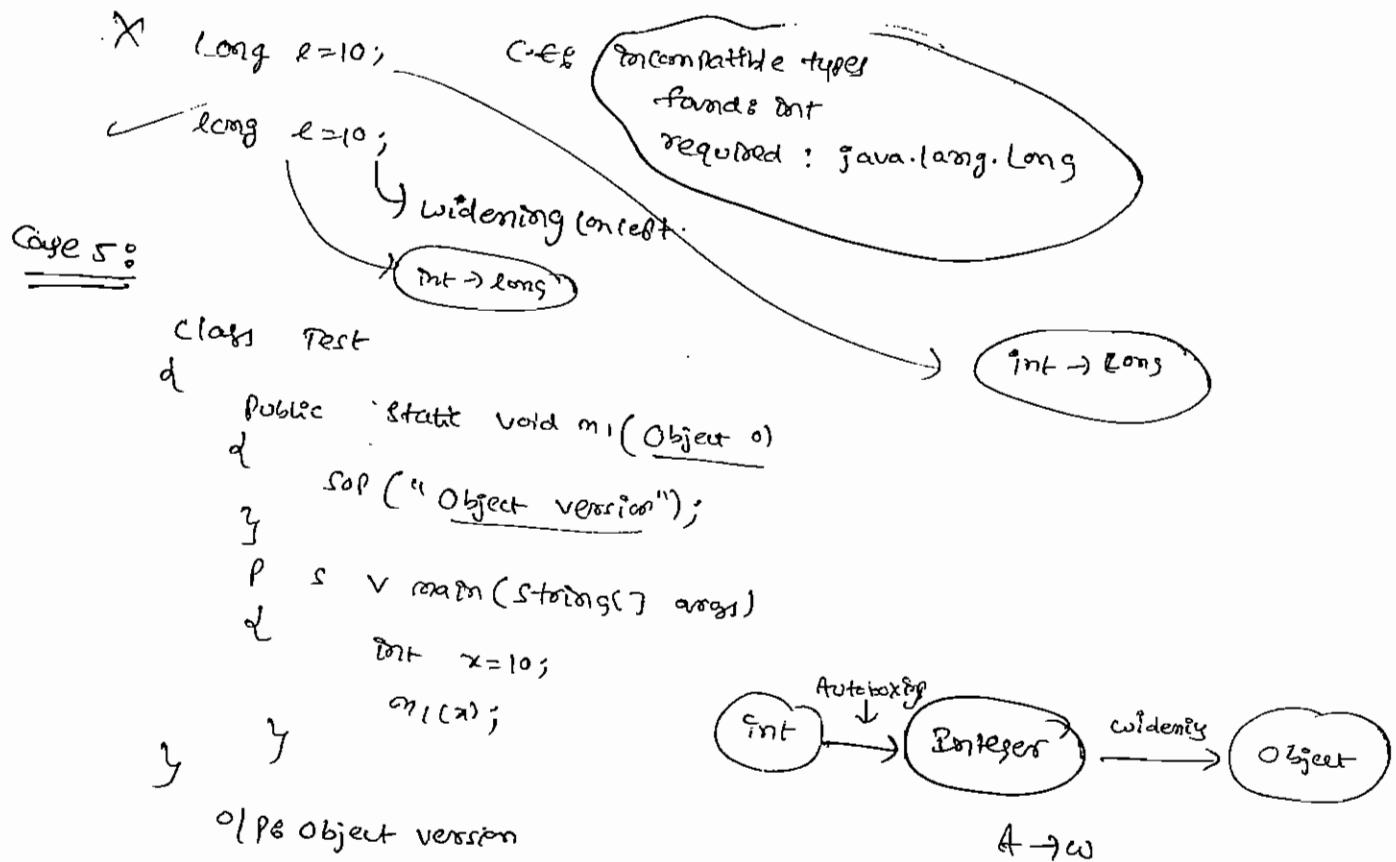
```

m1(i, long, Long) in Test

Can not be applied to int



widening followed by autoboxing is not allowed in Java where as autoboxing followed by widening is allowed



- which of the following assignments are legal
- `int i=10;`
  - `Integer I=10;` ✓ Autoboxing
  - `int q=10L;` ✓  $\text{Long} \rightarrow \text{int} \times$  C-E: (possible loss of precision found: long required: int)
  - `Long L=10L;` ✓ (Auto boxing)
  - `Long L=10;` ✗ ( $\text{int} \rightarrow \text{Long} \times$ ) C-E (incompatible types found: int required: Long)
  - `long l=10;` ✓ (widening)
  - `Object o=10;` ✓ (Autoboxing followed by widening)
  - `double d=10;` ✓ (widening)
  - `Double D=10;` ✗ ( $\text{int} \rightarrow \text{Double} \times$ ) C-E (incompatible types found: int required: Double)
  - `Number n=10;` ✓ ( $\text{Autoboxing} \rightarrow \text{widening}$ )

6

C

E

C

C

C

C

C

B

B

B

B

B

## File I/O

- (1) File
- (2) FileWriter
- (3) FileReader
- (4) BufferedWriter
- (5) BufferedReader
- (6) PrintWriter

File .

```
File f = new File ("abc.txt");
```

This line won't create any physical file first it will check if there any physical file named with abc.txt is available or not. If it is available then f simply refers that file. If it is not available then we are just creating java file object to represent the name abc.txt.

Ex:

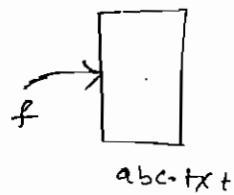
```
File f = new File ("abc.txt");
System.out.println (f.exists()); false
```

```
f.createNewFile();
```

```
System.out.println (f.exists()); true
```

is true

false	create
true	true



→ We can use java file object to represent directory also

Ex:

```
File f = new File ("durga23");
System.out.println (f.exists()); false
```

```
f.mkdir();
```

```
System.out.println (f.exists()); true
```



Note:

In UNIX everything is treated as a file Java File I/O concept is implemented based on UNIX operating system hence Java file object can be used to represent both files and directories.

1.0  
 1.1  
 1.2  
 1.3 } scjp ✓  
 7/8 a  
 PersistenceFil  
 1.4 } scjp x  
 1.5  
 1.6 } scjp ✓  
 1.7  
 count=0 26m 3  
 ②

dir abc.txt  
 search

## File class Constructors

① `File f = new File (String name);`

Creates a Java file object to represent name of the file or directory in current working directory

② `File f = new File (String subdirname, String name);`

Creates a java file object to represent name of the file or directory present in specified sub directory

③ `File f = new File (File subdir, String name);`

Same as ②

Ex 1: Write code to create a file named with abc.txt in current working directory?

```
File f = new File ("abc.txt");
f.createNewFile();
```

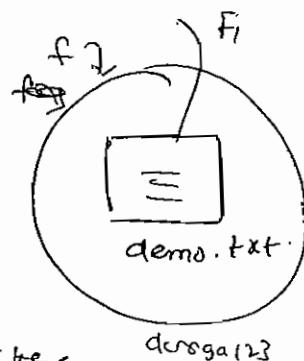


Ex 2: Write code to create a directory named with durga123 in current working directory, and create a file named with demo.txt in that directory.

```
file f = new File ("durga123");
f.mkdir();
```

```
// File fi = new File ("durga123", "demo.txt");
```

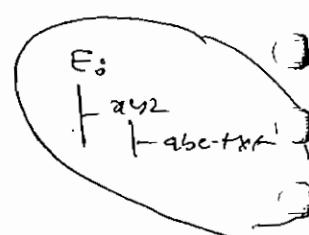
```
File fi = new File (f, "demo.txt");
fi.createNewFile();
```



Ex 3: Write code to create a file named with abc.txt in E:\xyz folder

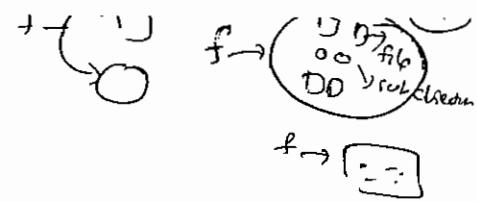
E:  
xyz

```
file f = new File ("E:\\xyz", "abc.txt");
f.createNewFile();
```



Assume that E:\xyz folder is already available in our system

## 2 important methods present in File



① `boolean exists();`

Returns true if the specified file or directory available

② `boolean createNewFile();`

first this method will check whether the specified file is already available or not. If it is already available then this method returns false without creating any physical file.

If the file is not already available then this method will creates a new file and returns true

③ `boolean mkdirs();`

④ `boolean isfile();`

Returns true if the specified file object pointing to physical file.

⑤ `boolean isDirectory();`

⑥ `String[] list();`

This method returns the name of all files and subdirectories present in specified directory

⑦ `long length();`

Returns number of characters present in the specified file

⑧ `boolean delete();`

To delete specified file or directory

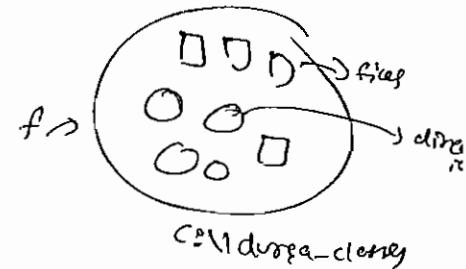
Ex: write a program to display the names of all files and directories present in `C:\durga-class`.

```
class test {
    public static void main(String[] args) throws Exception {
        int count = 0;
        file f = new file("C:\durga-class");
    }
}
```

`String[] s = f.list();`

```
for (String s1 : s) {
    count++;
    System.out.println(s1);
}
```

```
System.out.println("The total number: " + count);
```



Display only file names:

```

int count = 0;
File f = new File ("C:\\durga-classes");
String [] s = f.list();
for (String s1 : s)
{
    file f1 = new File (f1, s1);
    if (f1.isFile())
        count++;
    System.out.println(s1);
}
System.out.println ("The total no of files:" + count);

```

String class  
file method  
not file

```

if (s1.isFile())
{
    System.out.println(s1);
}

```

To Display only Directory names:

In the above program we have to replace ~~file~~ isFile() method with isDirectory() method.

(2)

## FileWriter

Ses-2

We can use FileWriter to write character data to the file.

### Constructors

① `FileWriter fw = new FileWriter (String fname);`

② `FileWriter fw = new FileWriter (File f);`



The above FileWriter meant for overriding of existing data instead of overriding if we want append operation then we have to create FileWriter by using the following constructors

③ `FileWriter fw = new FileWriter (String fname, boolean append);`

④ `FileWriter fw = new FileWriter (File f, boolean append);`

Note If the specified file is not already available then all the above constructors will create that file

## Methods

① **write(int ch)**

To write a single character

② **write(char[] ch)**

To write an array of characters

③ **write(String s)**

Bag nice

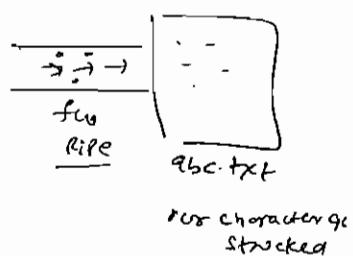
To write string to the file

④ **flush()**

To give the guarantee that total data including last character will be written to the file.

⑤ **close()**

To close the writer



for characters  
stacked

## Ex:

```
import java.io.*;
class FileWriterDemo2
{
    public static void main (String[] args) throws IOException
    {
        FileWriter fw = new FileWriter ("abc.txt");
        fw.write ('a'); // adding a single character
        fw.write (" Again Software Solutions ");
        fw.write ('\n');
        char [] ch = {'a', 'b', 'c'};
        fw.write (ch);
        fw.write ('\r');
        fw.flush ();
        fw.close ();
    }
}
```

q7 a  
q8 L  
q9 S  
q10 d

O/P:  
durga  
Software  
Solutions  
abc  
abc.txt

- \* In the above program `FileWriter` can perform Overriding of existing data instead of overriding if we want append operation then we have to create `FileWriter` object as follows

```
FileWriter fw = new FileWriter("abc.txt", true);
```

Note:

The main problem with `FileWriter` is we have to insert line separator (`\n`) manually which is varied from system to system so it is difficult to the programmer we can solve this problem by using BufferedWriter and PrintWriter classes

`\n` → newline character

durga\nsoftware\\solutions\\abc\\n...

\n

In some system `\n` not recognized as newline separator & treated as normal data

### (3) FileReader

We can use `FileReader` to read character data from file.

Constructors

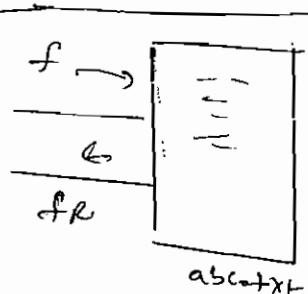
- ① `FileReader fr = new FileReader(String filename);`
- ② `FileReader fr = new FileReader(File f);`

Methods

- ① `int read()`
  - It attempts to read next character from the file and returning its unicode value
  - If the next character not available then this method returns -1
  - As this method returns unicode value (int value), at the time of printing we have to perform typecasting.

Exe

```
FileReader fr = new FileReader("abc.txt");
int i = fr.read();
while (i != -1)
    {
        System.out.print((char)i);
        i = fr.read();
    }
```



durga  
software  
solutions

abc.txt

int reads  
[unicode value  
char]  
int i = fr.read();  
for(i=100;  
i>

② int read(char[] ch)

It attempts to read enough characters from the file into char[] and returns number of characters copied from the file.



Exe

```
File f=new File ("abc.txt");
char[] ch=new char [(int)f.length()];
FileReader fr=new FileReader(f);
fr.read(ch);
```

for(char ch1:ch)  
{  
 sop(ch1);  
}

→ trying to read enough  
characters from file &  
Put into char[]

③ void close();

↳ after read/write recommended to use this method.

Exe

```
import java.io.*;
class FileReaderDemo
{
    public static void main(String[] args) throws IOException
    {
        File f=new File ("abc.txt");
        FileReader fr=new FileReader (f);
        char[] ch = new char [(int)f.length()];
        fr.read(ch);
        for(char ch1:ch)
        {
            sop(ch1);
        }
        sop("*****");
        FileReader fr1=new FileReader ("abc.txt");
        int i=fr1.read();
        while (i!= -1)
        {
            sop((char)i);
            i=fr1.read();
        }
    }
}
```

fr=new FileReader  
char[] ch=new char [100];  
fr.read(ch);

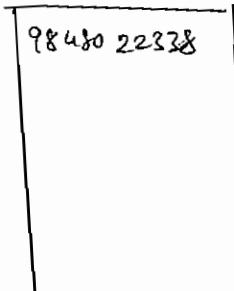
fr=new FileReader ("abc.txt");  
char[] ch=new char [100];  
fr.read(ch);

Maximum arrays  
int,  
buf.length() method  
return one long so  
true case  
not characters  
exceed if length  
first read

By using FileReader we can read data character by character which is not convenient to the programmer.

XXXXXX

Read



98450 22338

(to fetch mobile number)

Usage of FileWriter & FileReader → is not recommended because:

- ① while writing data by FileWriter we have to insert line separator (\n) manually which is varied from system to system it is difficult to the programmer
- ② By using FileReader we can read data character by character, which is not convenient to the programmer

To overcome these problems we should go for BufferedWriter and BufferedReader

#### ④ BufferedWriter

① we can use BufferedWriter to write character data to the file

Constructors

① BufferedWriter bw = new BufferedWriter(Writer w);

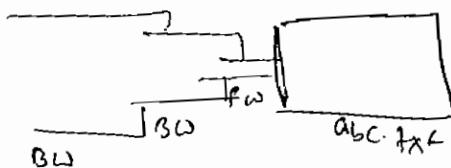
② BufferedWriter bw = new BufferedWriter(Writer w, int bufferSize);

Note: BufferedWriter can't communicate directly with the file  
it can communicate via some writer object.



Q) Which of the following are valid?

- ① BufferedWriter bw = new BufferedWriter("abc.txt");
- ② BufferedWriter bw = new BufferedWriter(new File("abc.txt")); 1) BufferedWriter can't communicate directly with the file
- ③ BufferedWriter bw = new BufferedWriter(new File("abc.txt")); file name or file object is same alike
- ④ BufferedWriter bw = new BufferedWriter(new BufferedWriter(new FileWriter("abc.txt")));



Two-level Buffering

⑥ line separator

## Methods

- ① write (int ch)
- ② writer (char[] ch)
- ③ write (String s)
- ④ flush()
- ⑤ close()
- \* ⑥ **newLine()**

To insert a line separator

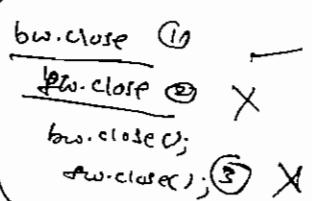
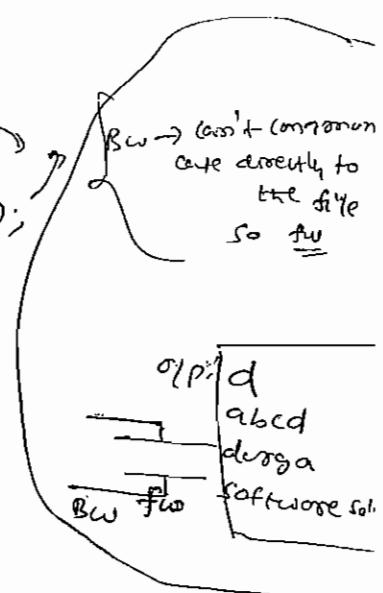
(Q) When compared with **FileWriter** which of the following capability available extra in method form ~~is~~ in **BufferedWriter**?

- ① writing data to the file
- ② close the file
- ③ ~~flushing the~~ flushing the file
- ④ inserting a new line character

## Exe

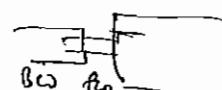
```

import java.io.*;
class BufferedWriterDemo
{
    public static void main(String[] args) throws IOException
    {
        FileWriter fw = new FileWriter ("abc.txt");
        BufferedWriter bw = new BufferedWriter (fw);
        bw.write ('a');
        bw.newLine();
        bw.write ('b');
        bw.newLine();
        bw.write ("durga");
        bw.newLine();
        bw.write ("Software Solutions");
        bw.flush();
        bw.close();
    }
}
  
```



Notes Whenever we are closing **BufferedWriter** automatically internal **FileWriter** will be closed and we are not required to close explicitly.

bw.close | fw.close | bw.close();  
✓ X X



## 5) BufferedReader

We can use BufferedReader to read character data from the file.

The main advantage of BufferedReader when compared with FileReader is we can read data line by line in addition to character by character.

### Constructors

① BufferedReader br = new BufferedReader(Reader r);

② BufferedReader br = new BufferedReader(Reader r, int bufferSize);

Note BufferedReader can't communicate directly with the file and it can communicate via some Reader object.

### methods:

① int read()

② int read(char[] ch)

③ void close()

④ String readLine()

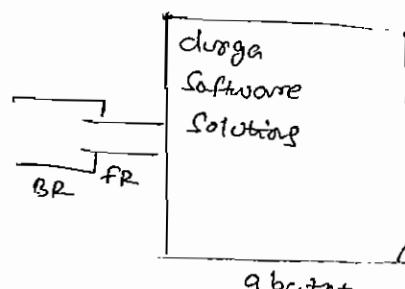
no next  
line  
now

It attempts to read next line from the file and returns it.  
If the next line not available then this method returns null.

Class BufferedReaderDemo

Ex: 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

when ever we are closing BufferedReader automatically underlying FileReader will be closed and we are not required to close explicitly.

### Note

The most enhanced Reader to read character data from the file is BufferedReader

## 16 PRINTWRITER

It is the most enhanced writer to write character data to the file.

The main advantage of PrintWriter over FileWriter is & BufferedWriter is we can write any type of primitive data directly to the file.

### Constructors

① PrintWriter pw = new PrintWriter (String fname);

② PrintWriter pw = new PrintWriter (File f);

③ PrintWriter pw = new PrintWriter (Writer w);

### Note:

PrintWriter can communicate directly with the file and can communicate via some Writer object also.

### Methods

① write (int ch)

② write (String s)

③ write (char[] ch)

④ flush ()

⑤ close ()

⑥ print (char ch);

⑦ print (int i);

⑧ print (double d);

⑨ print (boolean b);

⑩ print (String s);

⑪ println (char ch);

⑫ println (int i);

⑬ println (double d);

⑭ println (boolean b);

⑮ println (String s);

### Ex:

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

```
class PrintWriterDemo
```

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

```
    Filewriter fw = new FileWriter ("abc.txt");
```

```
    PrintWriter out = new PrintWriter (fw);
```

```
    out.write (100);
```

```
    out.println (100); → first print + then new line so  
    out.println (true); same line
```

```
    out.println ('c');
```

```
    out.println ("durga");
```

```
    out.flush ();
```

```
    out.close ();
```

(68) PW.out = new PrintWriter  
out = new PrintWriter

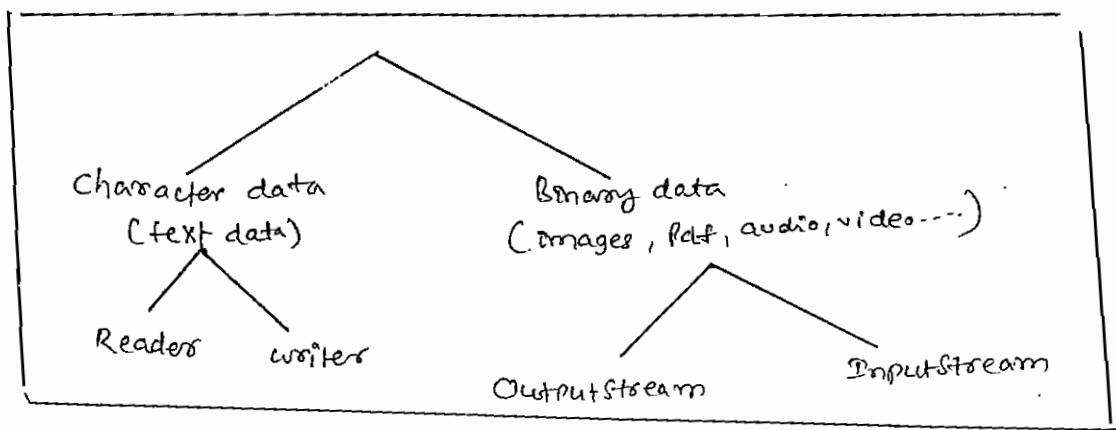
O/P:

```
durga  
true  
c  
durga
```

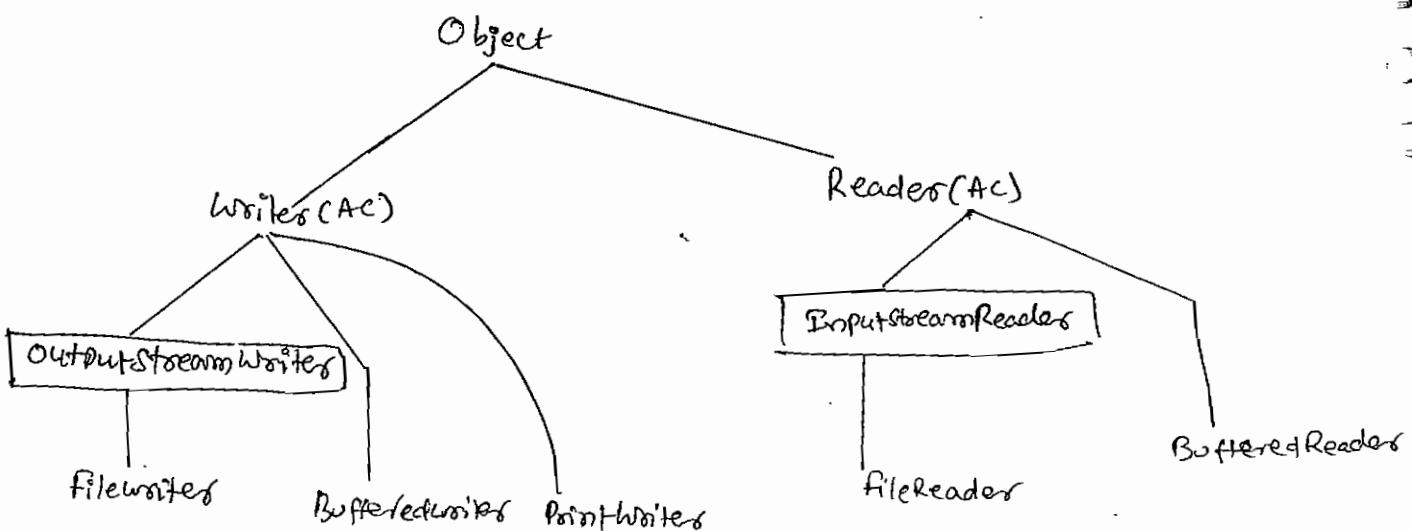
Q) what is the difference b/w `write(100)` and `print(100)`  
In the case of `write(100)` the corresponding character d will be added to the file but in the case of `print(100)` the int value 100 will be added to the file directly

Note: The most enhanced writer to write character data to the file is `PrintWriter` whereas the most enhanced reader to read character data from the file is `BufferedReader`

Note: In General we can use Readers & writers to handle character data (text data), whereas we <sup>can</sup> use streams to handle Binary data (like Images, Pdf files, video files, audio files etc--)



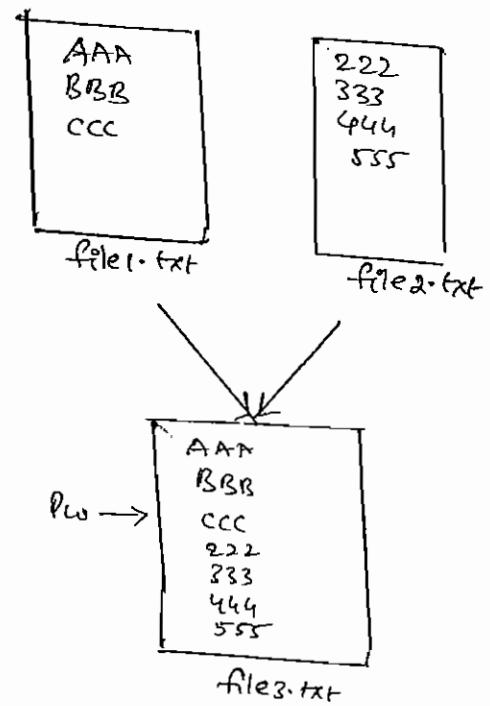
→ We can use OutputStream to write Binary data to the file, InputStream to read binary data from the file



(Q) write a program to merge data from two files into a third file?

```

import java.io.*;
class FileMerger
{
    public void main(String[] args) throws Exception
    {
        PrintWriter pw = new PrintWriter("file3.txt");
        BufferedReader br = new BufferedReader(new
            FileReader("file1.txt"));
        String line = br.readLine();
        while (line != null)
        {
            pw.println(line);
            line = br.readLine();
        }
        br = new BufferedReader(new FileReader("file2.txt"));
        line = br.readLine();
        while (line != null)
        {
            pw.println(line);
            line = br.readLine();
        }
        pw.flush();
        br.close();
        pw.close();
    }
}
  
```

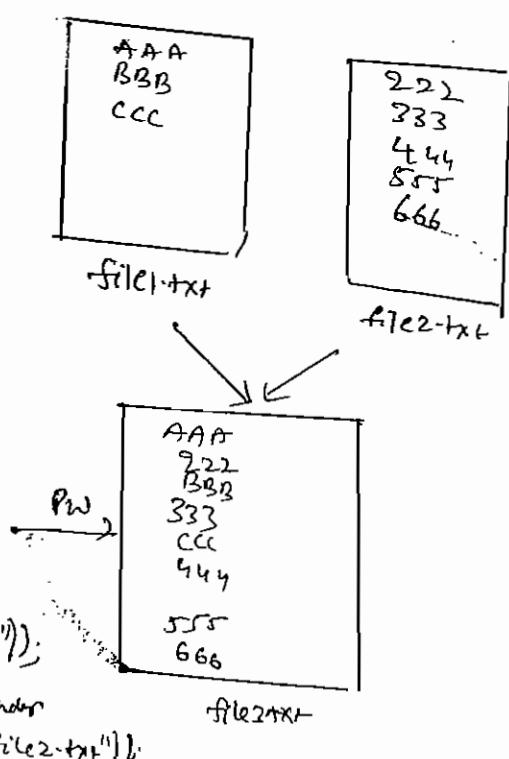


why not overriding  
we take only one  
PrintWriter if we  
take 2 Print write  
override)

(Q) write a program to perform file merge operation where merging should be done line by line alternatively  
here we take 2 Buffer readers to read simultaneously.

```

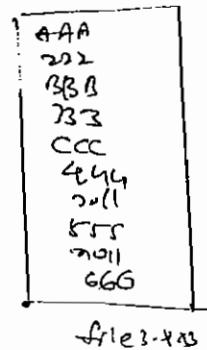
import java.io.*;
class FileMerger2
{
    public void main(String[] args) throws Exception
    {
        PrintWriter pw = new PrintWriter("file3.txt");
        BufferedReader br1 = new BufferedReader(new
            FileReader("file1.txt"));
        BufferedReader br2 = new BufferedReader(new FileReader("file2.txt"));
        pw.println(br1.readLine());
        pw.println(br2.readLine());
        pw.flush();
        br1.close();
        br2.close();
        pw.close();
    }
}
  
```



```

Storing line1 = br1.readLine();
Storing 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();
    }
}

```



(Q) Write a program to perform file extraction  
Operation?

Output = Input - delete

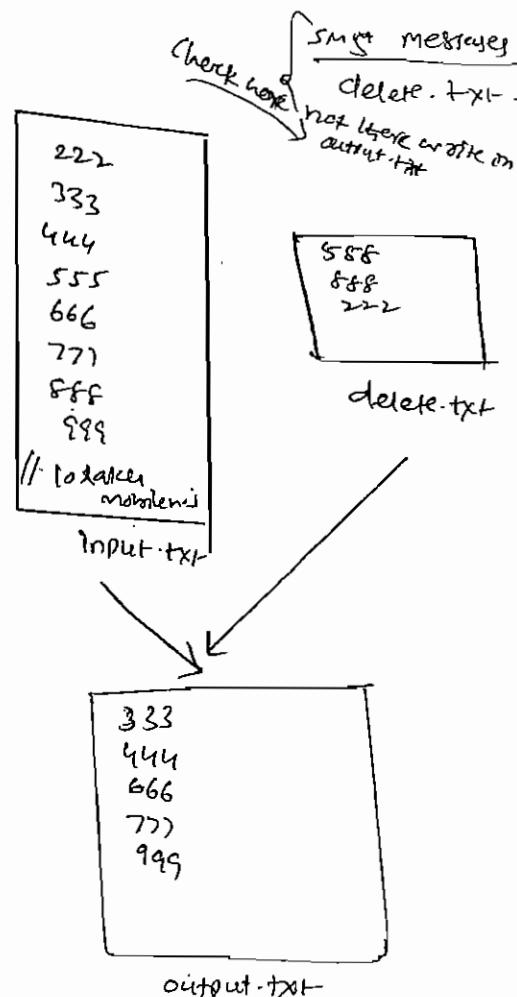
Read one number from input.txt & check if it is there in delete.txt if it is there in delete.txt not write in output.txt if it is not there write in output.txt.

e.g. SMG alert

To which file we have to write

Output.txt (②)

PrintWriter pw = new PrintWriter("output.txt");  
from input.txt we have to read  
by reading capability to use BufferedReader  
Br br = new Br (new Fr ("input.txt"));



```

import java.io.*;
class fileExtractor
{
    public static void main(String[] args) throws IOException
    {
        PrintWriter pw = new PrintWriter("output.txt");
        BufferedReader br1 = new BufferedReader("input.txt");
        String line = br1.readLine();
        while (line != null)
        {
            boolean available = false;
            BufferedReader br2 = new BufferedReader(new FileReader("delete.txt"));
            String target = br2.readLine();
            while (target != null)
            {
                if (line.equals(target))
                {
                    available = true;
                    break;
                }
                target = br2.readLine();
            }
            if (available == false)
            {
                pw.println(line);
            }
            line = br1.readLine(); // for all lines
        }
        pw.flush();
    }
}

```

222  
333  
444  
555  
666  
Input.txt

// compare every line in delete.txt  
 If match is not here  
 all values are  
 over write it  
 no to the  
 output.txt  
 cycle repeated for  
 every position  
 (Put br2 =  
 br1 = b)

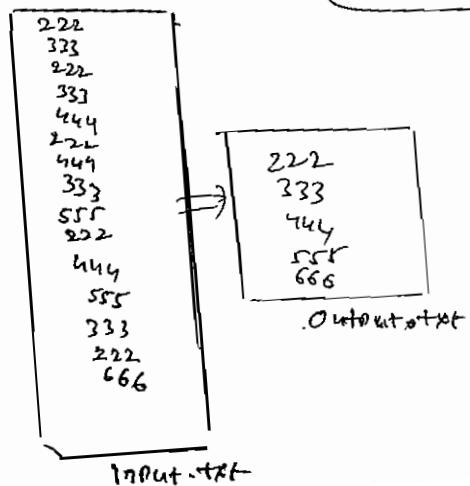
remove the duplicate lines & send excel

- Q) Write a java program to remove duplicates from the given input file?

Take number from input.txt

If it is there not write in ~~Output.txt~~  
 Output.txt if it is there not write  
 to output.txt

enquiry ->  
 demo  
 registration  
 adv java  
 sports  
 spring  
 ws  
 9 members  
 9 messages  
 we send  
 register  
 register  
 SMS com  
 attorney  
 ur in  
 project te



```

import java.util.*;
class DuplicateEliminator
{
    public static void main (String [] args) throws Exception
    {
        PrintWriter pw = new PrintWriter ("output.txt");
        BufferedReader br1 = new BufferedReader (new FileReader ("input.txt"));
        String line = br1.readLine ();
        while (line != null)
        {
            boolean available = false;
            BufferedReader br2 = new BufferedReader (new FileReader ("output.txt"));
            // Read output.txt to compare with input.txt
            String target = br2.readLine ();
            while (target != null)
            {
                if (line.equals (target))
                {
                    available = true; // If both are matched
                    break;
                }
                target = br2.readLine (); // If every line present
                // in output.txt
            }
            if (available == false)
            {
                // All numbers completed match is not there
                pw.println (line);
            }
            pw.flush ();
            line = br1.readLine (); // repeat total operation for
            // every line
        }
    }
}

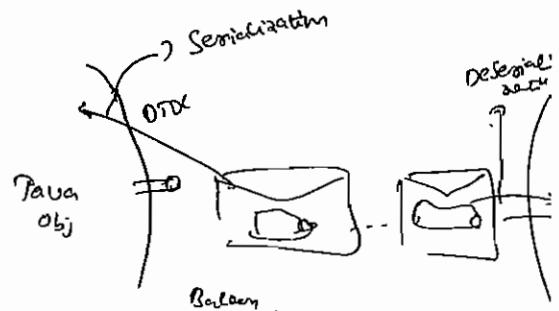
```

SCGP - 2013 Q

# Serialization

(b1)

- ① Introduction
- ② Object Graphs in Serialization
- ③ Customized Serialization
- ④ Serialization with respect to inheritance
- ⑤ Externalization
- ⑥ `SerialVersionUID`

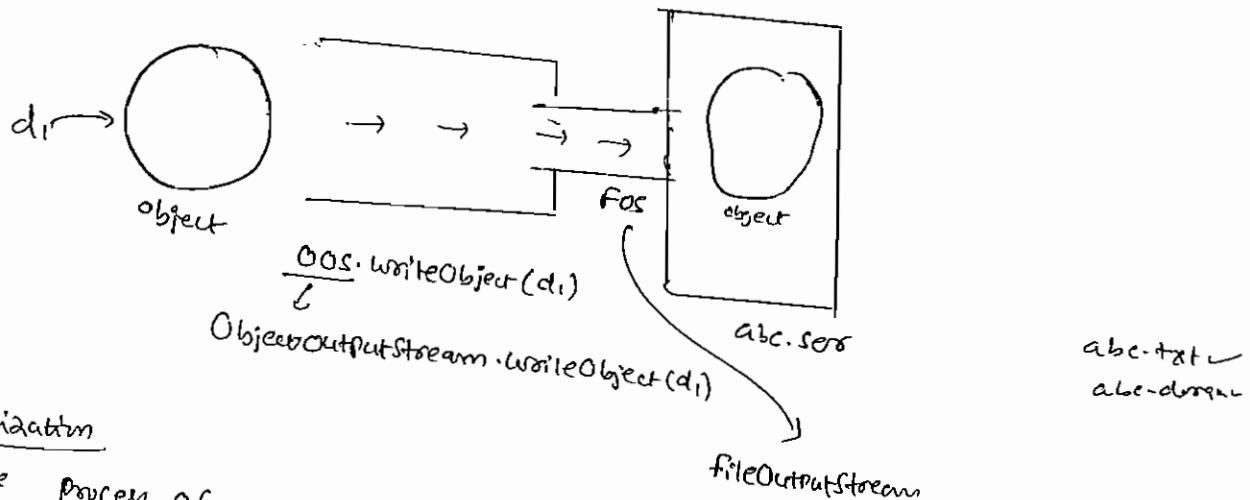


## Introduction

### Serialization

The process of writing state of an object to a file is called Serialization but strictly speaking it is the process of converting an object from Java supported form into either file supported form (or) network supported form.

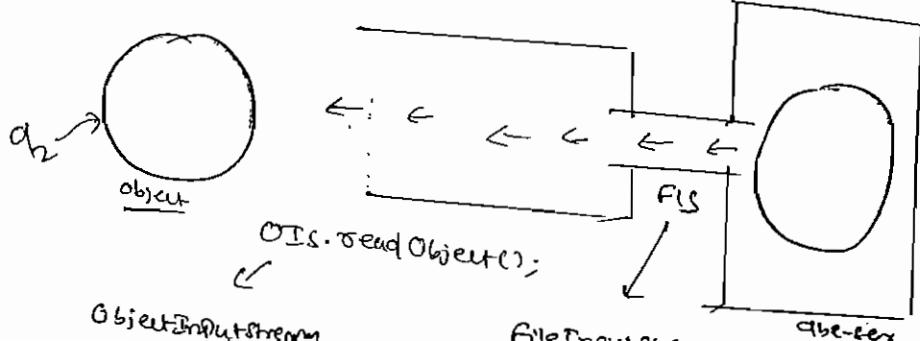
By using `FileOutputStream` and `ObjectOutputStream` classes we can achieve implementation of serialization.



### Deserialization

The process of reading state of an object from the file is called Deserialization but strictly speaking it is the process of converting an object from either file supported form or network supported form into Java supported form.

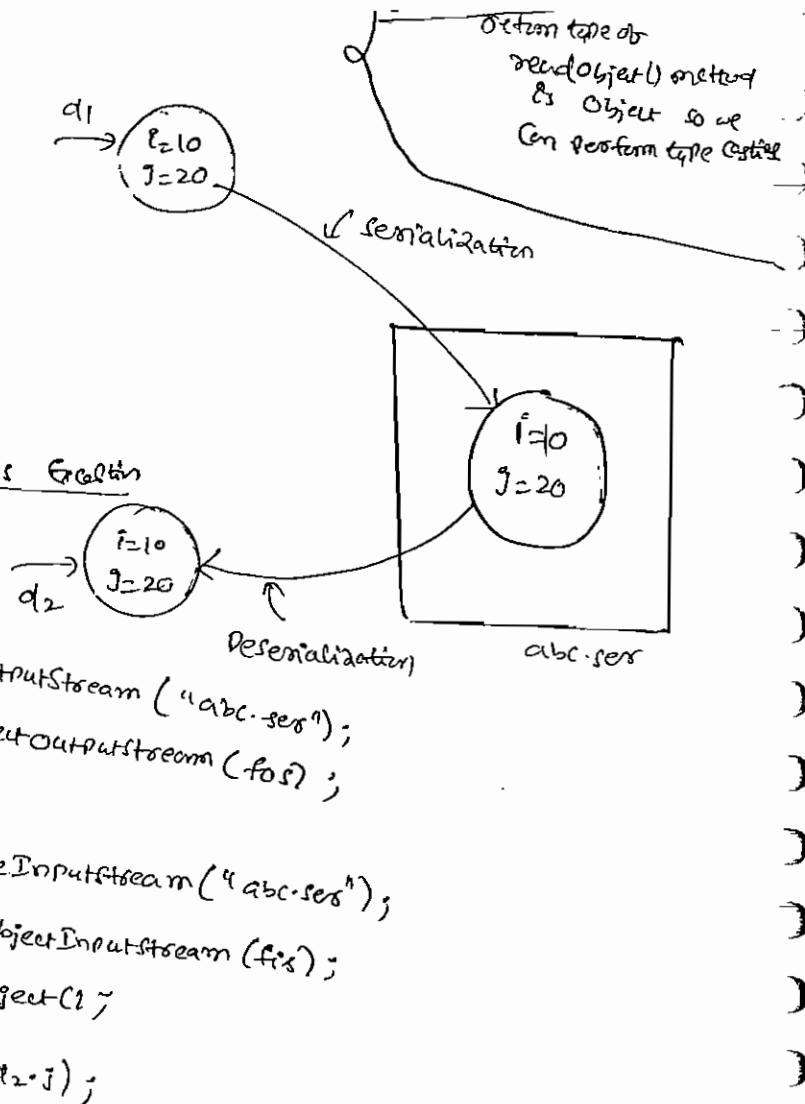
By using `FileInputStream` and `ObjectInputStream` classes we can implement deserialization.



```

Ex: import java.io.*;
class Dog implements Serializable {
    int i=10;
    int j=20;
}
class SerializeDemo {
    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();
        System.out.println("i=" + d2.i + " j=" + d2.j);
    }
}

```



\* We can serialize only serializable objects if and only if the corresponding class implements Serializable(?) interface.

Serializable(?) Interface present in `java.io` package and it doesn't contain any methods. It is a Marker Interface. If we are trying to serialize a non serializable object then we will get RuntimeException saying NotSerializableException.

### transient keyword

- transient modifier (keyword) applicable only for variables but not for methods and classes.
- At the time of serialization if we don't want to save the value of a particular variable to meet security constraints then we should declare that variable as transient.

(69)

while performing serialization JVM ignores the value of ~~original~~ transient variable and save default value to the file.

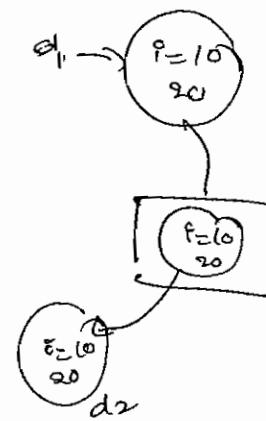
Hence transient means not to serialize.

### transient vs static

Static variable is not part of object state and hence it won't participate in serialization due to this declaring static variable as transient there is no use.

### final vs Transient

final variables will be participated in serialization directly by the value hence declaring a final variable as transient there is no impact.



j=20  
Static variable created first at the time of class loading  
Static variable not part of object but serialization class level for objects so static variable not participating in serialization + constant static  
use {not participating in serialization}

every final variable replaced with value at compilation only

```
final int x=10;
int y=20;
Sop(x); }= Sop(y)
```

Compile

at runtime final varia is not in variable then it is on Value

declaration	o/p
int i=10; int j=20;	10 --- 20
transient int i=10; int j=20;	0 ... 20
transient static int i=10; transient int j=20;	10 --- 0
transient int i=10; transient final int j=20;	0 ... 20
transient static int i=10; transient final int j=20;	(0 ... 20)

## Notes

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.  
i.e Order of Objects is important in serialization.

Exe

```
Dog d1 = new Dog();
```

```
Cat c1 = new Cat();
```

```
Rat r1 = new Rat();
```

```
fileOutputStream fos = new FileOutputStream("abc.ser");
```

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
oos.writeObject(d1);
```

```
oos.writeObject(c1);
```

```
oos.writeObject(r1);
```

```
FileInputStream fis = new FileInputStream("abc.ser");
```

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

```
Dog d2 = (Dog) ois.readObject();
```

```
Cat c2 = (Cat) ois.readObject();
```

```
Rat r2 = (Rat) ois.readObject();
```

If we don't know order of objects in serialization

Exe

```
FileInputStream fis = new FileInputStream("abc.ser");
```

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

```
Object o = ois.readObject();
```

```
if(o instanceof Dog)
```

```
Dog d2 = (Dog)o;
```

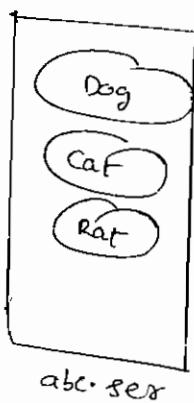
```
// Perform Dog specific functionality
```

```
else if(o instanceof Cat)
```

```
Cat c2 = (Cat)o;
```

```
// Perform Cat specific functionality
```

```
else if(o instanceof Rat)
```



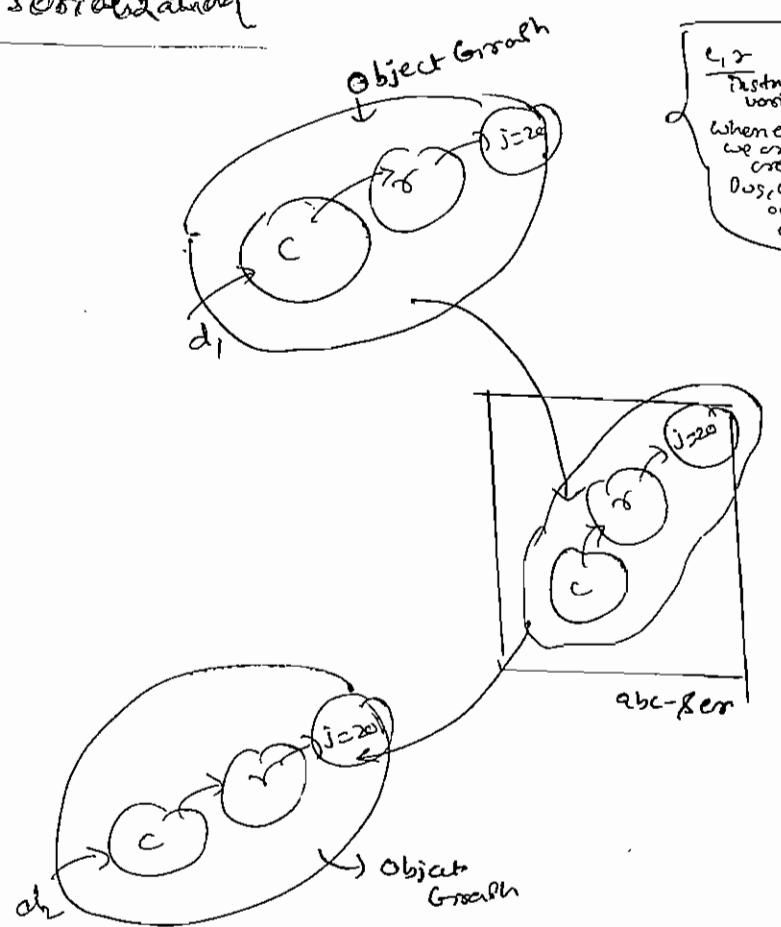
Object o

different reference  
can be used to  
hold any object  
reference

if it is the  
Dog type

## (2) Object Graphs in Serialization

(71)



e.g.  
Whenever we are creating Dog, Cat objects, one

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 Object Graph every object should be serializable if atleast one object is not serializable then we will get runtime exception NotSerializableException.

Exe

```
import java.io.*;
class Dog implements Serializable
{
    Cat c = new Cat();
}
class Cat implements Serializable
{
    Rat r = new Rat();
}
class Rat implements Serializable
{
    int j=20;
}
```

```

class SerializeDemo2
{
    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();
        System.out.println(d2);
    }
}

```

- In the above program whenever we are serializing Dog object automatically Cat and Rat objects got serialized because these are part of Object Graph of Dog.
- Among Dog, Cat and Rat Objects if atleast one object is not serializable then we will get Runtime Exception saying NotSerializableException

07/09/24

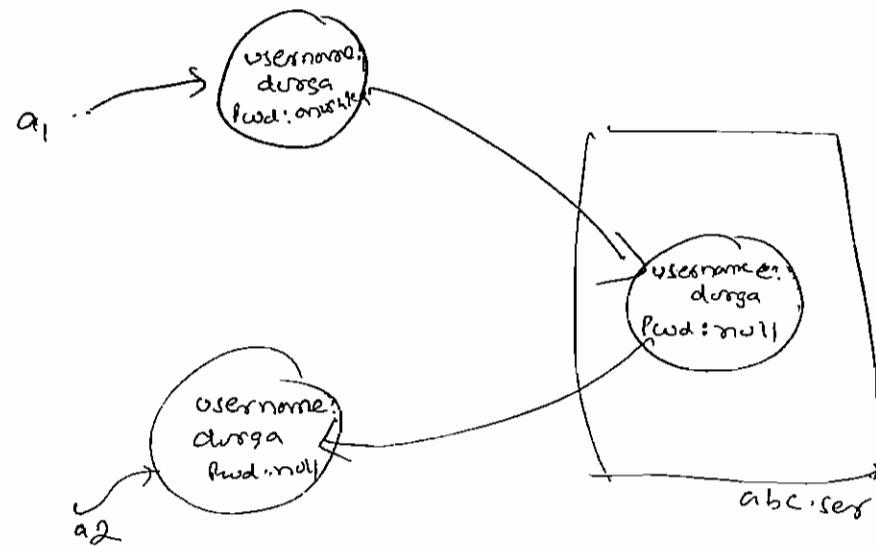
(73)

### ③ Customized Serialization

During default serialization there may be a chance of loss of information because of transient keyword.

Ex6

```
import java.io.*;
class Account implements Serializable
{
    String username = "durga";
    transient String Pwd = "amushka";
}
class CustSerializableDemo
{
    public static void main(String[] args) throws Exception
    {
        Account a1 = new Account();
        System.out.println(a1.username + " --- " + a1.Pwd); // durga -- amushka
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);
        System.out.println("Object written successfully");
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account) ois.readObject();
        System.out.println(a2.username + " --- " + a2.Pwd);
        ois.close();
    }
}
```



In the above example before serialization Account object can provide proper username & pwd but after deserialization Account object can provide only username but not password. This is due to declaring pwd variable as transient.

Hence during default serialization there may be a chance of loss of information because of transient keyword. To recover this loss of information we should go for customized serialization.

- → (1) Can implement customized serialization by using the following methods

(1) `private void writeObject(ObjectOutputStream os) throws Exception`

(2) If this method will be executed automatically at the time of serialization hence at the time of serialization if we want to perform any activity we have to define that in this method only.

(2) `private void readObject(ObjectInputStream is) throws Exception`

This method will be executed automatically at the time of deserialization hence at the time of deserialization if we want to perform any activity we have to define that in this method only.

#### Note

(1) The above methods are callback methods because these are executed automatically by the JVM (like interrupt)

(2) While performing which object serialization we have to do extra work in the corresponding class we have to define above methods. For example while performing Account object serialization if we required to do extra work in the account class we have to define above methods.

#### Ex:

```
import java.io.*;  
class Account implements Serializable
```

String username = "durga";

Transient String pwd = "amulshka";

FileOutputStream  
DataInputStream

Employee atm money frame office

OCM managing  
money

at sender side & do some extra work & receiver will do some extra work is called customized serialization

... One day another before encrypting

```

private void writeObject(ObjectOutputStream os) throws Exception
{
    os.defaultWriteObject(); // for addition to create method do default
    String ePwd = "123" + Pwd; // prepare encrypted password &
    os.writeObject(ePwd); // write into a file
}

private void readObject(ObjectInputStream is) throws Exception
{
    is.defaultReadObject();
    String ePwd = (String) is.readObject();
    Pwd = ePwd.substring(3);
}

class CustomizedSerializableDemo1
{
    public static void main(String[] args) throws IOException
    {
        Account a1 = new Account();
        System.out.println(a1.username + "----" + a1.Pwd); // durga---amushka.
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);
    }
}

```

$a_1 \rightarrow \text{username: durga Pwd: amushka}$

```

FileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
Account a2 = (Account) ois.readObject();
System.out.println(a2.username + "----" + a2.Pwd);
// durga---amushka

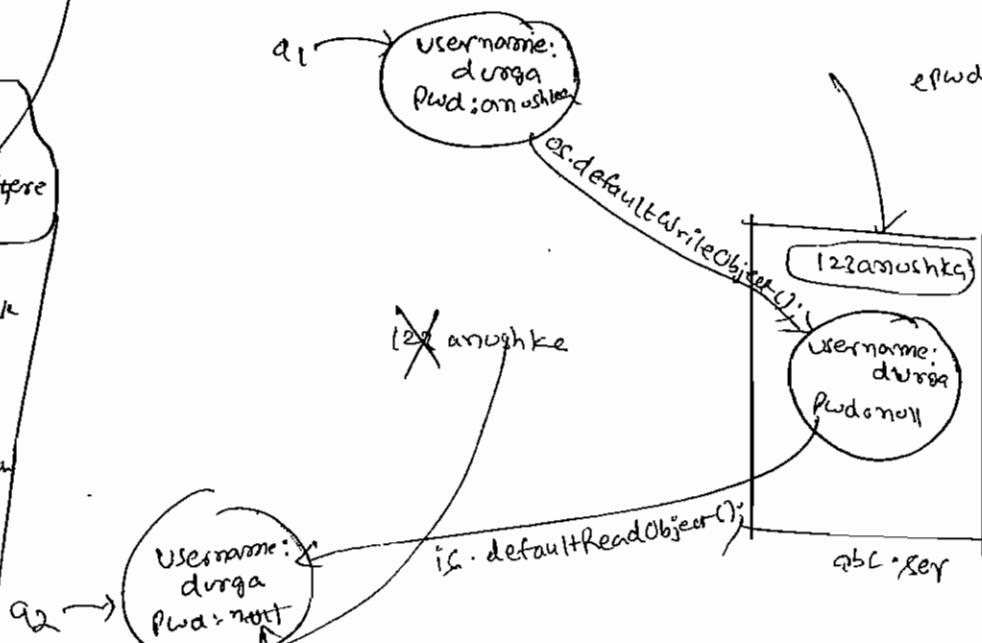
```

$a_2 \rightarrow \text{username: durga Pwd: amushka}$

here after serialization immediately run check  
Customized deserialization is there or not. if it is there  
then execute the method

Deserializable  
Immediately run check  
for Account class read  
object method is there or  
not, Yes if it is there  
it do default deserialization  
Then read encrypted  
password & decrypt it  
if index to end

At the time of  
deserialization run  
will check if there  
any write object method



On the above programs before serialization and after serialization

Account Object can provide proper username and password.

### Notes

Programmer can't call private methods directly from outside of the class. but JVM can call private methods directly from outside of the class.

### Example

```
import java.io.*;  
class Account implements Serializable  
{  
    String username = "durga";  
    transient String pwd = "anushka";  
    transient int pin = 1234;  
    private void writeObject(ObjectOutputStream os) throws Exception  
    {  
        OS.defaultWriteObject();  
        String epwd = pwd + pin;  
        int epin = 4444 + pin;  
        OS.writeObject(epwd);  
        OS.writeInt(epin);  
    }  
    private void readObject(ObjectInputStream is) throws Exception  
    {  
        OS.defaultReadObject();  
        String epwd = (String) is.readObject();  
        pwd = epwd.substring(0);  
        int epin = is.readInt();  
        pin = epin - 4444;  
    }  
}  
class SerializeDemo2  
{  
    public static void main(String[] args) throws Exception  
    {  
        Account a1 = new Account();  
        System.out.println(a1.username + " " + a1.pwd + " " + a1.pin);  
        FileOutputStream fos = new FileOutputStream("a.abc.ser");  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        oos.writeObject(a1);  
    }  
}
```

transient  
pwd  
pin  
secured  
If more than  
one transient  
variable is there  
how you can  
handle

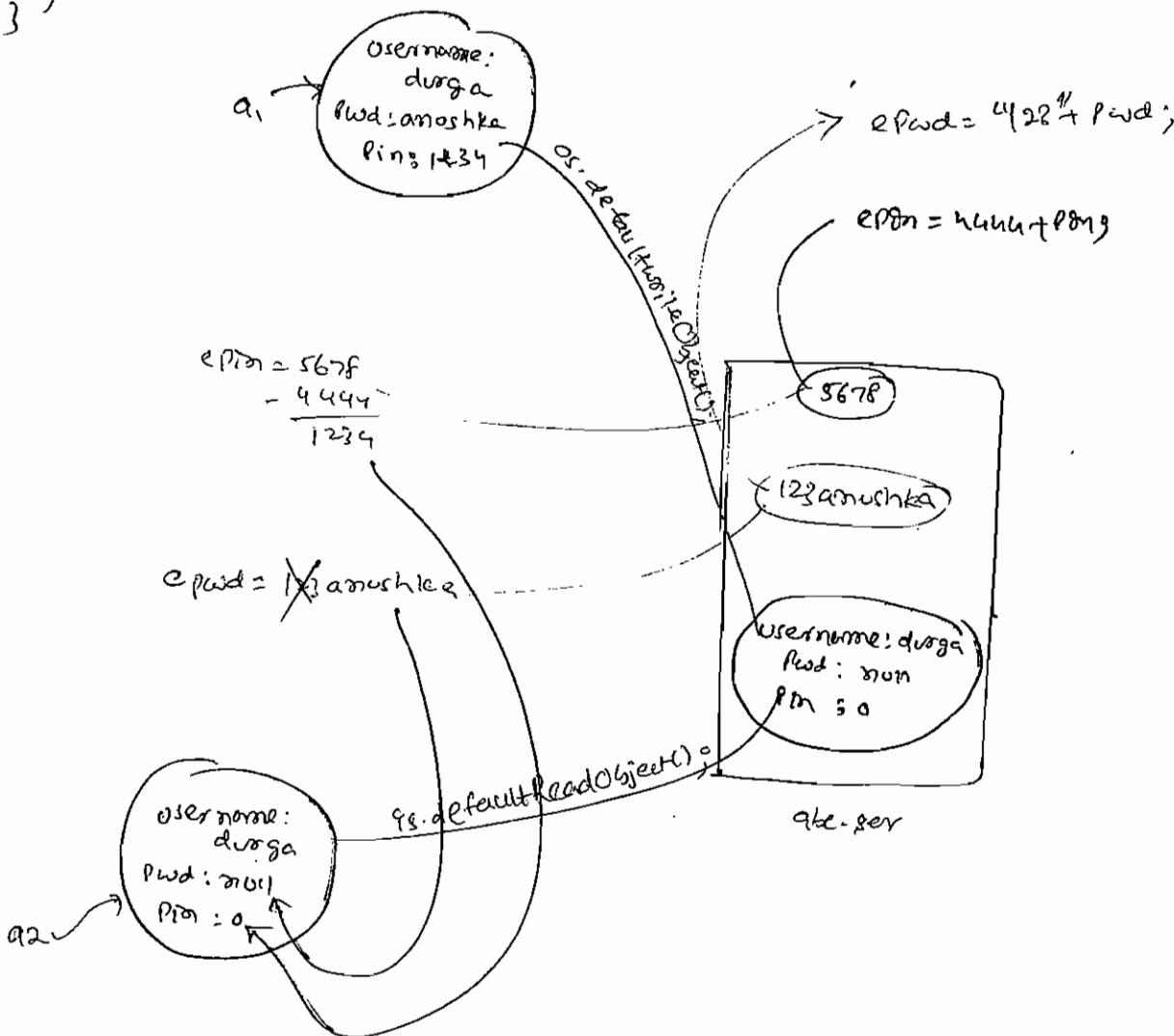
```

fileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
Account a2 = (Account) ois.readObject();
for (a2.getUsername() + " - " + a2.getPassword() + " - " + a2.getPin());
}
}

```

(7)

ans  
ex  
56



(4)

#### Serialization with respect to Inheritance

Case 1:

Even though child class doesn't implement `Serializable` we can serialize child class object if parent class implements `Serializable` interface. Hence if parent is inheriting from parent to child `Serializable`.

Ex:

```

import java.io.*;
class Animal implements Serializable {
    int i=10;
}
class Dog extends Animal {
    int j=20;
}
class

```

## L14) Serializable Example

→ Public static void main (String [] args) throws exception

    { Dog d1 = new Dog ();

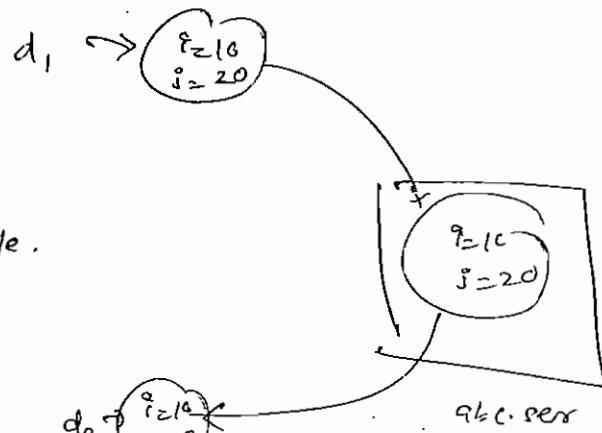
        System.out.println (d1.i + " " + d1.j);

    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();  
    System.out.println (d2.i + " " + d2.j);

}

In the above example even though  
Dog class doesn't implement  
Serializable we can serialize  
Dog object because it's Parent  
Animal class implements Serializable.



## Notes

Object class doesn't implement

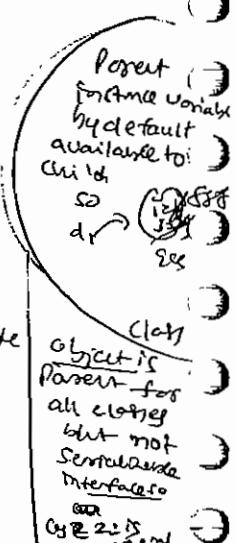
Serializable Interface.

## Case 2:

→ ① Even though Parent class doesn't implement Serializable we can serialize child class object if child class implements Serializable interface i.e To serialize child class object Parent class need not be Serializable.

→ ② At the time of serialization JVM will check is any variable inheriting from non Serializable Parent or not if any variable inheriting from non Serializable Parent then JVM ignores original value and save default value to the file.

③ At the time of Deserialization JVM will check is any Parent class nonSerializable (or) not. if any Parent class is non Serializable then JVM will execute instance control flow in every non Serializable Parent and share it's instance variable to the current object.

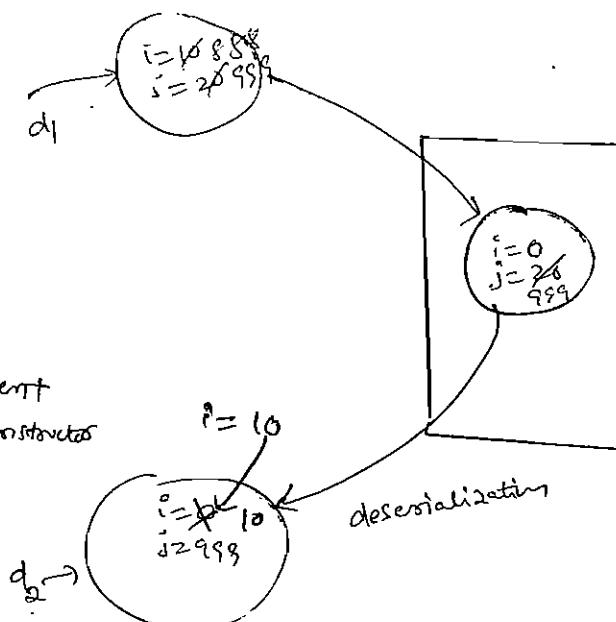


④ while executing instance

Control flow of non serializable

Parent JVM will always call no-argument constructor hence every non-serializable class should compulsorily contain no-argument constructor it may be default constructor generated by compiler (or) customized constructor explicitly provided by programmer.

otherwise we will get Runtime Exception saying InvalidClassException.



instance control flow

- 1) Initialization of instance members
- 2) Execution of instance variable assignments & instance blocks
- 3) Execution of const

→ while creating object  
constructors called

1) Direct  
2) Indirect  
JVM will execute non-serializable  
instance control flow  
called to  
Animal contr  
called  
every non serializable  
parameter

```

import java.io.*;
class Animal {
    int i = 10;
    Animal() {
        System.out.println("Animal constructor called");
    }
}
class Dog extends Animal implements Serializable {
    int j = 20;
    Dog() {
        System.out.println("Dog constructor called");
    }
}
class SerializeDemo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        d1.i = 888;
        d1.j = 999;
    }
}

```

```

fileOutputStream fos = new FileOutputStream("abc.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(d1);
System.out.println("Deserialization started");
fileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
Dog d2 = (Dog) ois.readObject();
System.out.println(d2.i + " " + d2.j);

```

O/P's

Animal Constructor Called  
Dog Constructor Called  
Deserialization Started  
Animal Constructor Called  
(0----999)

if we don't constructor  
for the parent class  
constructor compiler default  
provide no arg constructor

## Externalization

- ① In serialization everything takes care by JVM and programmer doesn't have ~~any~~ any control.
- ② In serialization it is always possible to save total object to the file and it is not possible to save part of the object, which may creates performance problems

To overcome these problems we should go for externalization.

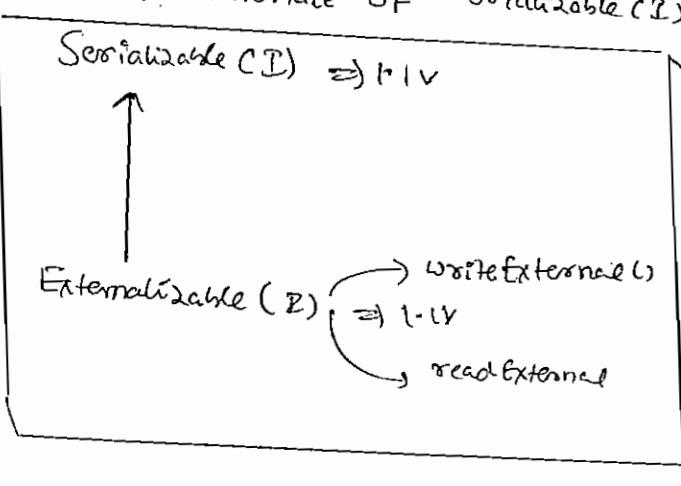
- ③ The main advantage of externalization over serialization is ~~base~~ everything takes care by programmer and JVM doesn't have any control.  
Based on our requirement we can save either total object or part of the object, which improves performance of the system.

To provide externalizable ability for any Java object compulsorily the corresponding class should implement Externalizable interface.

Externalizable(I) interface defines 2 methods

- ① writeExternal()
- ② readExternal()

→ Externalizable(I) is the child interface of Serializable(I)



① public void writeExternal(ObjectOutput out) throws IOException

This method will be executed automatically at the time of Serialization  
With in this method we have to write code to save required variables to the file.

② public void ReadExternal(ObjectInput in) throws IOException, ClassNotFoundException

This method will be executed automatically at the time of Deserialization  
With in this method we have to write code to read required variables from the file and assign to current Object.

\* → But strictly speaking at the time of Deserialization Jvm will create a separate new Object by executing Public no-argument constructor on that Object Jvm will call `readExternal()` method.

\* → Hence every Externalizable implemented class should compulsorily contain Public no-argument constructor, otherwise we will get RuntimeException saying InvalidClassException

EX  
import java.io.\*;  
public class ExternalizableDemo  
implements Externalizable

```

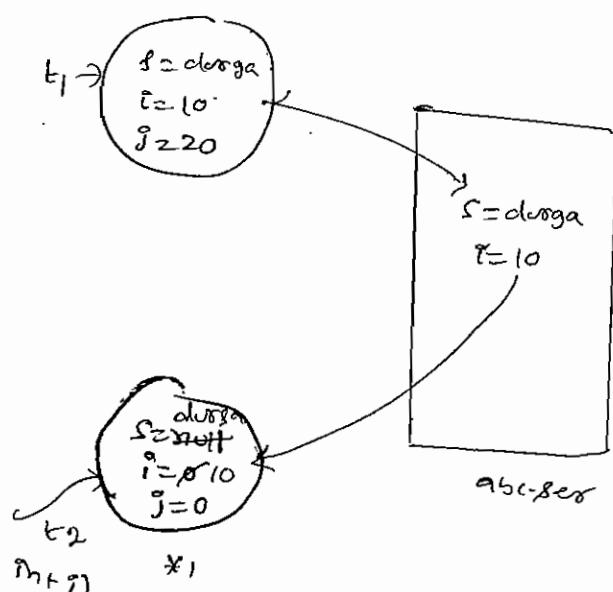
    String s;
    int i;
    int j;
    public ExternalizableDemo()
    {
        System.out.println("Public no-arg constructor");
    }
    public ExternalizableDemo (String s, int i, int j)
    {
        this.s=s;
        this.i=i;
        this.j=j;
    }

```

```

    public void writeExternal (ObjectOutput out) throws IOException
    {
        out.writeObject(s);
        out.writeInt(i);
    }

```



```

public void readExternal (ObjectInput in) throws IOException, ClassNotFoundException {
    s = (String) in.readObject();
    i = in.readInt();
}

public static void main (String [] args) throws Exception {
    ExternalizableDemo t1 = new ExternalizableDemo ("durga", 1020);
    FileOutputStream fos = new FileOutputStream ("abc.ser");
    ObjectOutputStream oos = new ObjectOutputStream (fos);
    oos.writeObject (t1);
}

fileInputStream fis = new FileInputStream ("abc.ser");
ObjectInputStream ois = new ObjectInputStream (fis);
ExternalizableDemo t2 = (ExternalizableDemo) ois.readObject();
System.out.println (t2.s + " --- " + t2.i + " --- " + t2.o);
}

```

→ If the class implements Serializable then total object will be saved to the file in this case output is O/P = durga..10...20

If the class implements Externalizable then only required variables will be saved to the file in this case output is O/P = { public no-arg constructor } { durga --- 10 --- 0 }

#### Note :-

In Serialization transient keyword will play role but in Externalization transient keyword won't play any role of course transient keyword not required in Externalization.

implements  
Serialization  
& But  
Externalizable  
abc.ser strategy

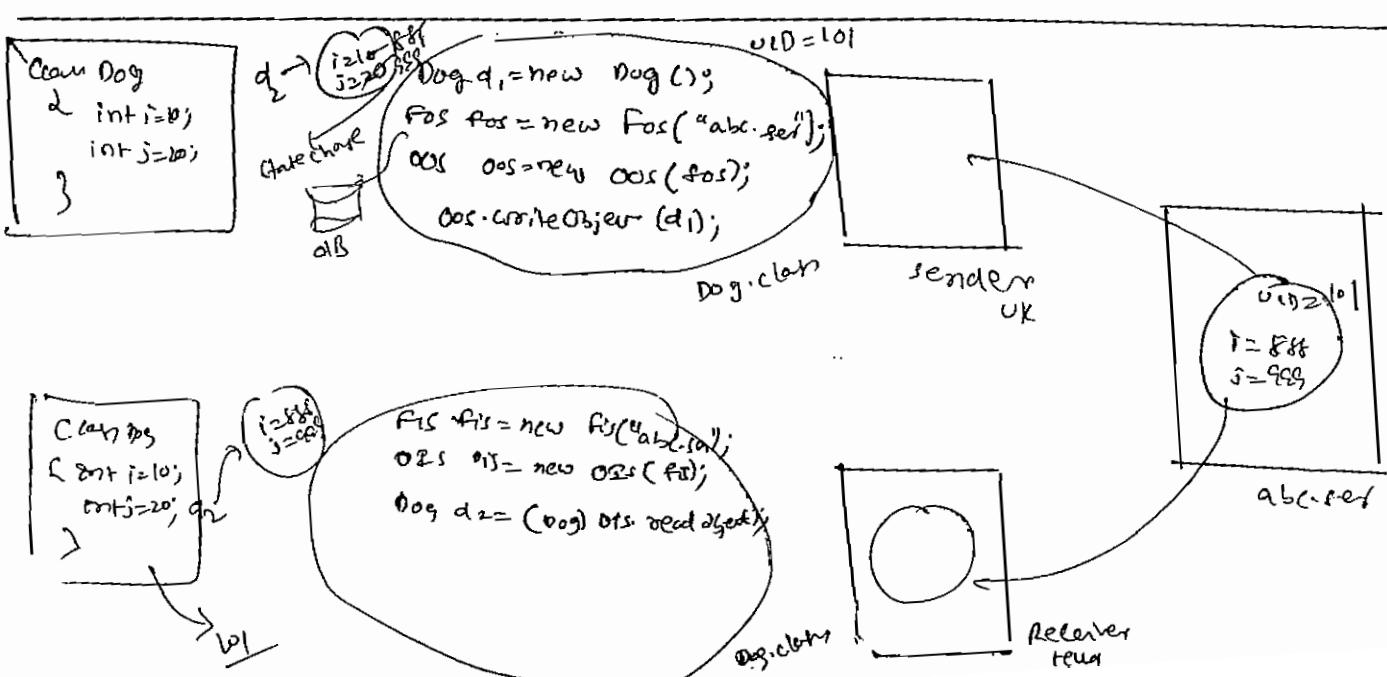
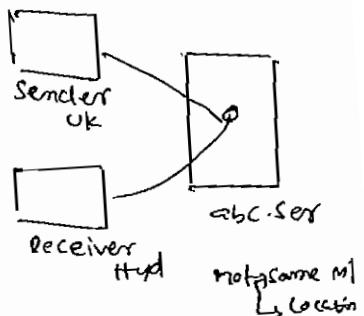
#### Difference b/w Serialization and Externalization :

Serialization	Externalization
① It is meant for default serialization	① It is meant for customized serialization
② Here everything takes care by JVM and programmer doesn't have any control	② Here everything takes care by programmer and JVM doesn't have any control
③ In this case it is always possible to save total object to the file and it is not possible to save part of the object.	③ Based on our requirement we can save either total object or part of the object

- (4) Relatively performance is low
- (5) It is the best choice if we want to save total object to the file
- (6) Serializable interface doesn't contain any methods and it is a marker interface
- (7) Serializable implemented class not required to contain public no-argument constructor
- (8) transient keyword will play role in serialization
- (9) Relatively performance is high.
- (10) It is the best choice if we want to save part of the object to the file
- (11) Externalizable interface contains (2) methods (1) writeExternal() & (2) readExternal()
- and hence it is not a marker interface
- (12) Externalizable implemented class should compulsorily contain public no-arg constructor otherwise we will get RuntimeException saying InvalidClassException
- (13) transient keyword won't play any role in externalization, of course it won't be required.

### 7) SerialVersionUID

- \* In serialization both Sender and Receiver need not be same person, need not to use same machine and need not be from the same location. The persons may be different, the machines may be different and locations may be different.
- \* In serialization Both sender and receiver should have .class file at the beginning only just state of object is travelling from sender to receiver



At the time of serialization with every object senderside JVM will save a unique identifier.

JVM is responsible to generate this unique identifier based on .class file at the time of deserialization receiverside JVM will compare this unique identifier associated with the object with local class unique identifier if both are matched then only deserialization will be performed otherwise we will get Runtime exception saying InvalidClassException.  
→ This unique identifier is nothing but SerialVersionUID.

Problems of Depending on Default SerialVersionUID generated by JVM.

- (1) Both sender and Receiver should use same JVM with respect to vendor and platform and version otherwise receiver unable to deserialize because of different SerialVersionUID's.
- (2) Both sender and receiver should use same .class file version. after serialization if there is any change in .class file at receiver side then receiver unable to deserialize.
- (3) To generate SerialVersionUID internally JVM may use complex algorithm which may create performance problems.

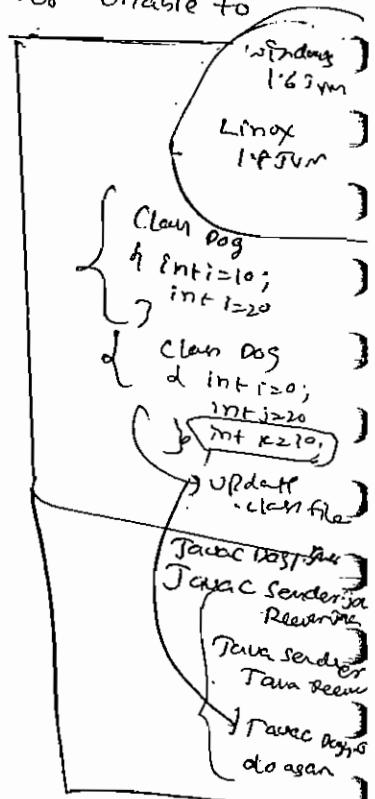
We can solve above problems by configuring our own SerialVersionUID

We can configure our own SerialVersionUID as follows

```
private static final long serialVersionUID = 1L;
```

Ex:

```
import java.io.*;
class Dog implements Serializable
{
    private static final long serialVersionUID = 1L;
    int i=10;
    int j=20;
```



```

import java.io.*;
class Sender
{
    P = void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
    }
}

```

### Receiver.java

```

import java.io.*;
class Receiver
{
    P = void main(String[] args) throws Exception
    {
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog) ois.readObject();
        System.out.println("d2.i = " + d2.i);
    }
}

```

In the above program after serialization if we perform any change to the class file at receiver side we won't get any problem at time of Deserialization.

In this case Sender and Receiver not required to maintain same JVM versions.

#### Note:

- ① Some IDE's explicitly prompt programmer to enter serialVersionUID
- ② Some IDE's may generate serialVersionUID automatically.

13

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

□

04/08/2014

## Collection framework

(87)

- An array is an indexed collection of fixed number of homogeneous data elements.
- The main advantage of arrays is we can represent multiple value by using single variable. So that readability of the code will be improved.

### Limitation of Object type Arrays

- 1) Arrays are fixed in size
  - ie once we created an array there is no chance of increasing (or) decreasing array size.
- 2) Arrays can hold only homogeneous data elements

```
Student[] s = new Student[1000];
```

```
s[0] = new student();
```

```
s[1] = new student();
```

```
s[2] = new customer(); X ✓
```

- 3) That is an array can hold only same type of objects by mistake if we are trying to add another type of object then we will get compile time error.

CaE  
In incompatible type  
found: Customer  
Required: student

- 4) We can overcome this limitation by using Object[] (Object array)

```
Object[] a = new Object[1000];
```

```
a[0] = new student(); ✓
```

```
a[1] = new customer(); ✓
```

- 5) There is no ready made data support available for arrays for every requirement we should write the logic explicitly so that complexity of the programming will be increased.

To overcome the above limitations we should go for Collections

The main advantages of Collections are

- 1) Collections are auto growable in nature i.e based on our requirement we can increase or decrease size of the collections
- 2) Collections can hold both homogeneous and heterogeneous objects
- 3) Every Collection class internally implemented with the support of data structures so that we can expect ready made data support for Collections.

Ques) differences between Arrays and Collections

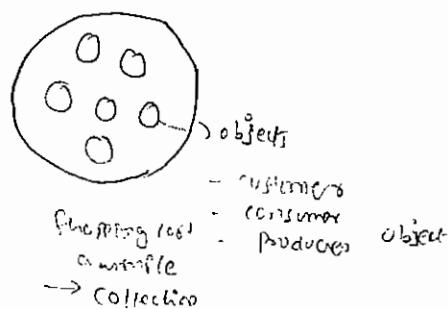
	Arrays	Collections
1)	Arrays are fixed in size	1) Collections are autogrowable in nature
2)	With respect to memory arrays are not recommended	2) With respect to memory collections are recommended
3)	With respect to performance arrays are recommended	3) With respect to performance collections are not recommended. Collections support inheritance data structures so it's logic for creating data structures to increase the performance.
4)	Applicable for both primitives & objects $\text{Object} \& \text{int}[]$ - int[] objects - String[]	4) Applicable only for objects but not for primitives $\text{Object} \& \text{String}$
5)	Arrays can hold only homogeneous elements. (except Object[]) Object array	5) Collections can hold both homogeneous and heterogeneous objects.
6)	There is no underline data structure support and hence we can't expect ready made data support for arrays	6) Every collection class internally implemented with the support of data structure and hence we can expect ready made data support for collections

## Collection

If we want to represent a group of individual objects as a single entity then we should go for Collection.

## Collection framework

It defines several classes and interfaces to represent a group of individual objects as a single entity



# 9 Key interfaces of Collection framework

(89)

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap

[05/08/14]

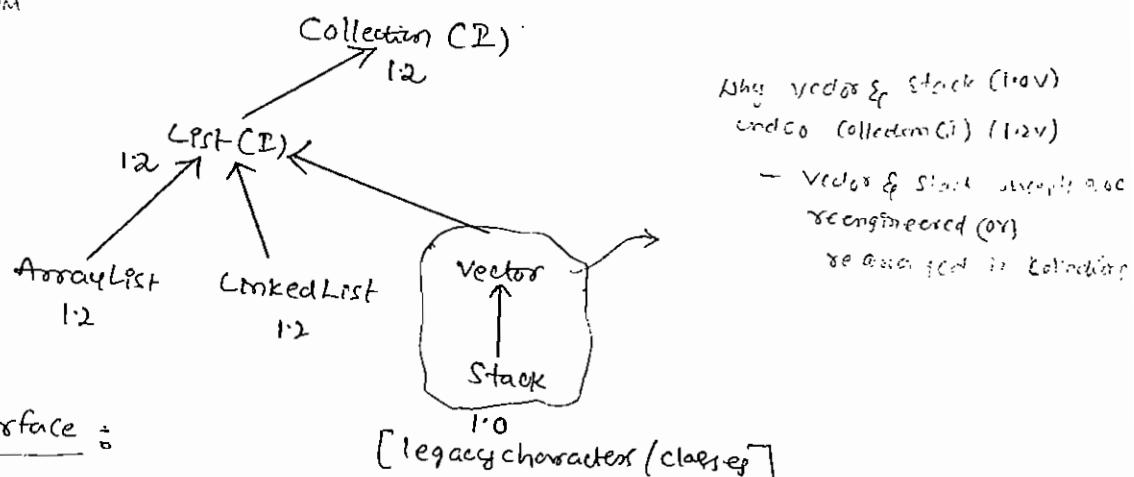
## 1) Collection interface :

- If we want to represent a group of individual objects as a single entity then we should go for Collection interface.
- It acts as root for Collection framework
- It defines the most common methods that are applicable for any Collection object
- There is no class which is directly implementing Collection interface.

## 2) List interface : (Duplicates + insertion orders Preserved)

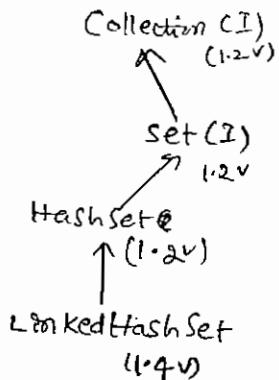
- It is the child interface of Collection
- If we want to represent a group of individual Objects as a single entity where duplicates are allowed & insertion orders will be preserved then we should go for List interface.

what are the ways to preserve  
u insert the object  
the same place in the JVM  
already



## 3) Set interface :

- It is the child interface of Collection
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion orders won't be preserved then we should go for Set interface.



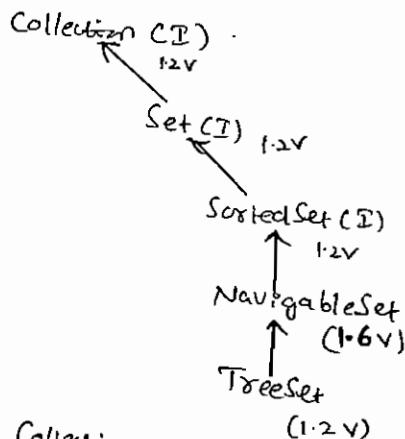
#### 4) SortedSet interface

- It is the child interface of Set
- If we want to represent a group of individual objects as a single entity with out duplicates and according to some sorting order then we should go for SortedSet interface.

#### 5) NavigableSet interface

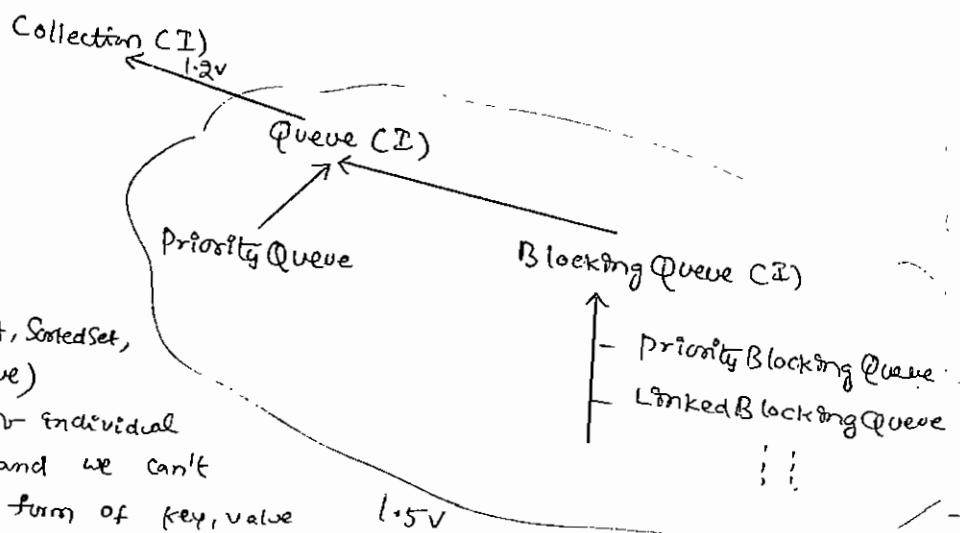
- It is the child interface of SortedSet
- It defines several methods for navigation purpose

100, 200, 300, 400, 500



#### 6) Queue interface

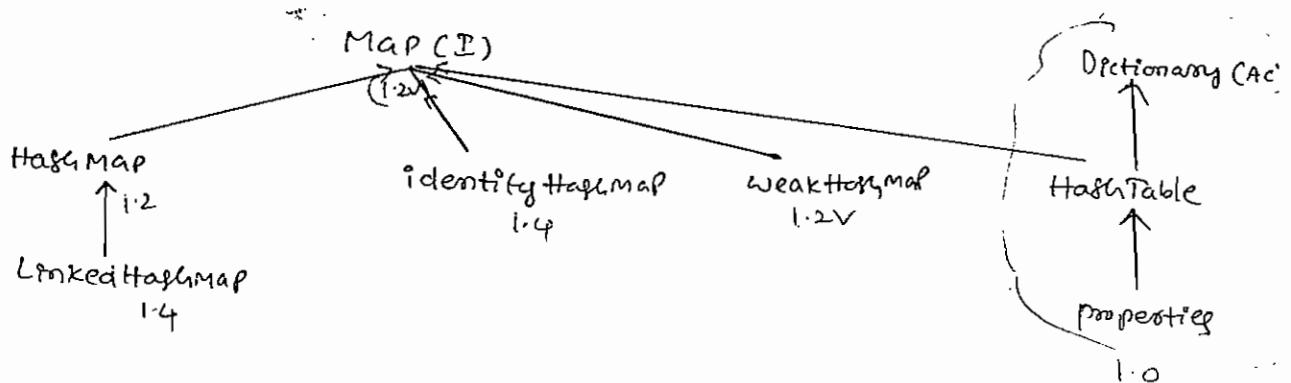
- It is the child interface of Collection
  - If we want to represent a group or individual objects prior to processing then we should go for Queue interface.
- FIFO representation



→ By using all the above interfaces (Collection, List, Set, SortedSet, NavigableSet & Queue) we can represent a group or individual objects as a single entity and we can't use to represent in the form of key, value pairs.

## 7) Map interface

- If we want to represent a group of Objects as key value pairs then we should go for Map interface.
- Map is not child interface of Collection

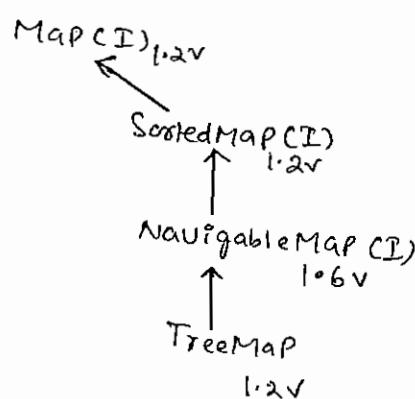


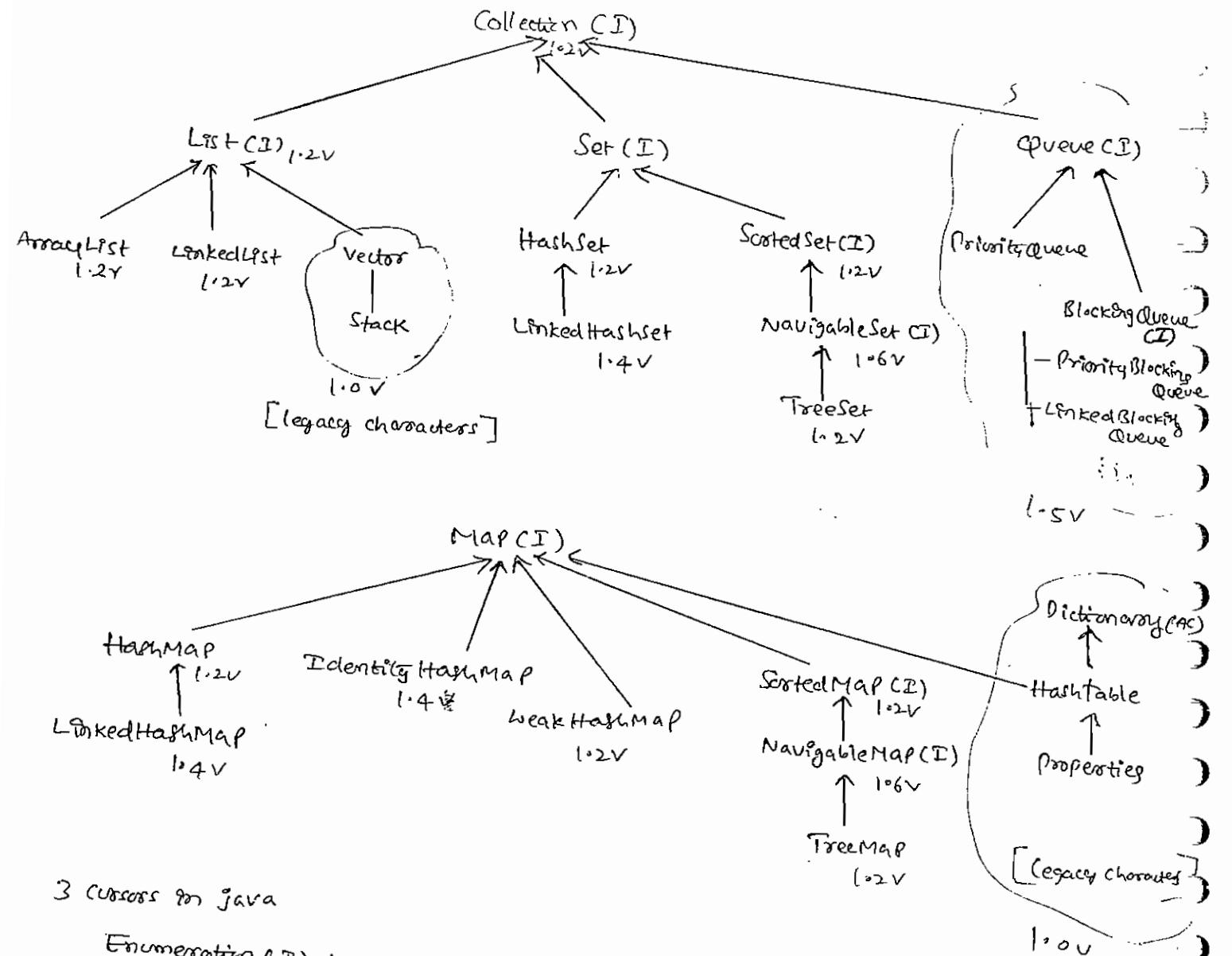
## 8) SortedMap

- It is the child interface of Map
- If we want to represent a group of Objects as key, value pairs and according to some sorting order of keys then we should go for SortedMap.

## 9) NavigableMap

- It is the child interface of SortedMap
- it defines several methods for navigation purpose





### 3 Cursors in java

Enumeration(I) 1.0V  
 Iterator(I) 1.2V  
 ListIterator(I) 1.2V

### Sorting Purpose

Comparable(I) 1.2V → Dnso (Default natural sorting order)  
 Comparator(I) 1.2V → cso (Customized sorting order)

### Utility classes

Collections 1.2  
 Arrays 1.2

(legacy characters 1.0V)

Enumeration(I)  
 Dictionary (AC)  
 Hashtable  
 Properties  
 Vector  
 Stack

Collection (I) :

- ① If we want to represent a group of individual objects of a single entity then we should go for Collection interface.
  - ② It acts as root for Collection framework.
  - ③ It defines the most common methods that are applicable for any Collection object.
  - ④ There is no class which directly implements Collection interface.
- methods

```

boolean add(Object o)
boolean addAll(Collection c)
boolean remove (Object o)
boolean removeAll (Collection c)
boolean retainAll (Collection c)

```

To remove all objects except those present in c.

```

boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
void clear()
int size()
Object[] toArray();
Iterator iterator();

```

There is no getter method in Collection interface to get required object.

Collection vs Collections

- Collection is an interface which can be used to represent a group of individual objects of a single entity.
- Collections is a utility class which defines several utility methods that are applicable for Collection objects.

## List(I) :

- It is the child interface of Collection
- If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order will be preserved then we should go for List interface
- We can preserve insertion order and we can differentiate duplicates by means of index / by using index with the List.

### List Interface methods

- void add(int index, Object o)
- boolean addAll(int index, Collection c)
- Object get(int index)
- Object remove(int index)
- Object set(int index, Object newObj)
  - to replace the elements present at specified index with provided Object and returns an old object
- int indexOf(Object o)
  - Returns the index of first occurrence of 'o'
- int lastIndexOf(Object o)
  - Returns the index of last occurrence of 'o'
- ListIterator listIterator()

### (a) ArrayList

- 1) The underlined datastructure is growable array (or) resizable array
- 2) duplicates are allowed
- 3) insertion orders will be preserved
- 4) Heterogeneous objects are allowed.
  - (except TreeSet & TreeMap everywhere heterogeneous objects are allowed)
- 5) Null insertion is possible

### Constructors

① `ArrayList l = new ArrayList();`

Creates an empty ArrayList object with default initial capacity 16.

If ArrayList reaches its max capacity then it will be resized automatically with

$$\boxed{\text{New Capacity} = (\text{currentCapacity} * \frac{3}{2}) + 1}$$

② `ArrayList l = new ArrayList(int initialCapacity);`

Creates an empty ArrayList with the specified initialCapacity.

③ `ArrayList l = new ArrayList(Collection c);`

Creates an equivalent ArrayList object for the given Collection object.

This constructor meant for interconversion between Collection objects.

### Example 1:

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        l.add("Pawan");
        l.add(100);
        l.add("Samantha");
        l.add(10.5);
        l.add(null);
        l.add(100);
    }
    System.out.println(l);
}
```

O/P: [ Pawan, 100, Samantha, 10.5, null, 100 ]

### Example 2:

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
    }
}
```

```

l.add("Srikanth");
l.add(123);
l.add("Shwathi");
l.add(null);
l.add(2);
System.out.println(l); [Srikanth, 123, Shwathi, null, 2]
l.remove(2); object
System.out.println(l); [Srikanth, 123, null, 2].
l.add("M");
l.add(2, "N");
System.out.println(l); [Srikanth, 123, N, null, 2, M]
}

```

O/P

→ Usually we can use Collection Object to hold data and to transfer data from one system to another system across the network.

- to provide support for this requirement Every Collection class implements Serializable and Cloneable interface (java.io package)
  - ArrayList and Vector classes implements RandomAccess Interface also, so that we can access any element randomly with the same retrieval time. (first, last, random element or last element) (first, last, random element or last element)
- RandomAccess Interface Present in java.util package and it doesn't contain any methods it is a marker interface.

Ex6

```

ArrayList l= new ArrayList();
System.out.println(l instanceof Serializable); O/P // true
System.out.println(l instanceof Cloneable); // true
System.out.println(l instanceof RandomAccess); // true

```

- But linkedList doesn't implement RandomAccess interface.

```

LinkedList l= new LinkedList();
System.out.println(l instanceof Serializable); O/P // true
System.out.println(l instanceof Cloneable); // true
System.out.println(l instanceof RandomAccess); // false

```

remove(index) - Collection remove(int index) - list

↳ remove(index)

↳ It is going to remove  
↳ an element and  
↳ end of the list

ArrayList	Vector
① No method present inside ArrayList is synchronized.	① Every method present inside Vector is synchronized.
② Multiple threads are allowed to operate ArrayList object simultaneously and hence it is not threadsafe.	② Only one thread is allowed to operate Vector object and hence it is thread safe.
③ Threads are not required to wait so that relatively performance is high.	③ Threads are required to wait so that relatively performance is low.
④ Introduced in 1.2v and it is not legacy.	④ Introduced in 1.0v & it is a legacy.

- ArrayList is the best choice if our frequent requirement is retrieval operation, but it is the worst choice if our frequent requirement is either insertion or deletion of the objects in the middle because it internally required several shift operations.
- If our frequent requirement is either insertion or deletion in the middle then we should go for LinkedList.

07-05-2014

### LinkedList (I)

- The underline datastructure is DoublyLinkedList
- Duplicates are allowed
- Insertion order will be preserved
- Heterogeneous objects are allowed.
- Null insertion is possible
- LinkedList class implements both Serializable and Cloneable Interfaces but it doesn't implement RandomAccess interface.

#### Constructors

- ① `LinkedList l = new LinkedList();`
- ② `LinkedList l = new LinkedList(Collection c);`

- usually we can use `LinkedList` to implement Stacks & Queues.
- to provide support for this requirement `LinkedList` class defines the following specific methods.

```

void addFirst(Object o)
void addLast(Object o)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()
    
```

Eg:

```

import java.util.*;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList l = new LinkedList();
        l.add("durga");
        l.add(30);
        l.add(null);
        l.add("durga");
        l.set(0, "Software");
        l.add(0, "Venky");
        System.out.println(l.getLast()); // durga
        l.removeLast();
        l.addFirst("ccc");
        System.out.println(l); // [ccc, Venky, Software, 30, null]
    }
}
    
```

ccc, venky, software, 30, null

*represents Queue or stack*

*to replicate*

- `LinkedList` is best choice if our frequent operation is either insertion or deletion of objects in the middle.

## Vector

- The underline datastructure is growable array or resizable array
- duplicate objects are allowed
- insertion order will be preserved
- heterogeneous objects are allowed
- null insertion is possible

Vector class implements

- Vector is the best choice if our frequent operation is retrieval operation.
- every method present in vector is by default synchronized hence Vector object is thread safe.

## methods:

To add the objects

`add(Object o) --> Collection(I)`

`add(int index, Object o) --> List(I)`

`addElement(Object o) --> Vector`

To remove the objects

`remove(Object o) --> Collection`

`removeElement(Object o) --> Vector`

`remove(int index) --> List`

`removeElementAt(int index) --> Vector`

`clear() --> Collection`

`removeAllElements() --> Vector`

To retrieve the elements

`Object get(int index) --> List`

`Object elementAt(int index) --> Vector`

`Object firstElement() --> Vector`

`Object lastElement() --> Vector`

## Some other methods

`int size()`

`int capacity()`

`Enumeration elements()` // to retrieve the elements

## Examples

### Constructors

① `Vector v = new Vector();`

Creates an empty vector object with default initial capacity 10  
Once if vector object reaches its max capacity then automatically it will be resized with

$$\boxed{\text{new capacity} = 2 * \text{current capacity}}$$

② `Vector v = new Vector(int initialCapacity);`

Creates an empty vector object with the specified initial capacity

`add(Object o) --> ?` (79)  
`add(int index, Object o) --> ?`  
`add(): ? (Object o) --> ?`  
`remove(): ?`  
`remove(Collection c) --> ?`  
`remove(int index) --> ?`  
`elementAt(int index) --> ?`  
 Capacity: how many objects can be held  
 200  
`size() --> ?`  
 how many objects are held  
 100  
 Generalization  
 Contains  
 Contains Elements

③ Vector v = new Vector(int initialCapacity, int incrementalCapacity);

Vector v = new Vector(Collection c);

### Examples

```
import java.util.*;  
class VectorDemo  
{  
    public static void main(String[] args)  
    {  
        Vector v = new Vector();  
        System.out.println(v.capacity()); // 10  
        for (int i=0; i<10; i++)  
        {  
            v.addElement(i);  
        }  
        System.out.println(v.capacity()); // 10  
        v.addElement("markant");  
        System.out.println(v.capacity()); // 120  
        System.out.println(v); [1, 2, 3, 4, 5, 6, 7, 8, 9, markant]  
    }  
}
```

→ vector is the best choice if our frequent operation is retrieval but it is the worst choice if our frequent operation is either insertion or deletion of the objects in the middle because it required several shift operations.

### Stack

- It is the child class of vector
- It is a data structure for First In Last Out order (FIFO)

#### Constructor

Stack s = new Stack();  
Creates an empty stack object

#### methods

Object push (Object o)

to insert an object into the stack

Object pop ()

to remove and return top of the stack

Object peek()

to return top of the stack without removal

Ctrl + Z last  
moving file delete  
ctrl + C copy

Vector methods are  
also available for  
Stack also.

{  
 Pop → remove &  
 return  
 Peek → return  
 Top → return  
 clear → empty  
 add → insert  
 remove → delete  
}

Object, int capacity

{ } offset

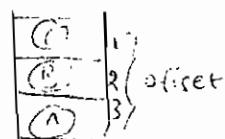
`boolean empty()`  
Return true if stack is empty

`int search(Object o)`

return offset if the element is available otherwise returns -

### Example:

```
import java.util.*;  
  
class StackDemo  
{  
    public static void main(String[] args)  
    {  
        Stack s=new Stack();  
        System.out.println(s.empty()); // true  
        s.push("A");  
        s.push("B");  
        s.push("C");  
        System.out.println(s); // [A, B, C] → we are not changing the elements  
        System.out.println(s.search("A")); // 0 return offset  
        System.out.println(s.search("Z")); // -1  
    }  
}
```



for all methods of class  
just up till the character here  
increasing order is maintained in stack  
[A, B, C]

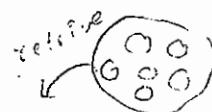
(08-08-2014)

### The 3 Cursors of Java

We can use cursors to retrieve the elements one by one from Collection object.

The following are the 3 cursors available in Java

- 1) Enumeration (I) (1.0v)
- 2) Iterator (I) [1.2v]
- 3) ListIterator (I) (1.2v)



### 1) Enumeration (I)

We can use Enumeration cursor to retrieve the elements one by one from Collection object.

- We can get Enumeration cursor by using elements() method of Vector class

`[public Enumeration elements();]`

Ex: `Enumeration e = v.elements();`

vector object

(we can not create object for interface)  
[Serializable obj]  
[Collection obj]  
[Iterator interface  
in Collection class]  
Dynamic interface

class A{  
 private  
 ↓ =  
3 class is called it

↓  
3 class is called it

B is an interface

- Once we get Vector object we can call the following methods on that object

```
public boolean hasMoreElements();
public Object nextElement();
```

Ex:

```
import java.util.*;
class EnumerationDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        for (int i=0; i<10; i++)
        {
            v.addElement(i);
        }
        System.out.println(v); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    }
}
```

```
// Enumeration e = v.elements();
// System.out.println(e.getNext());
// while (e.hasMoreElements())
// {
```

```
    Integer I = (Integer)e.nextElement();
    if (I%2==0)
```

```
{
```

```
    System.out.println(I);
}
}
System.out.println(v); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Vector v = new Vector();
Enumeration e = v.elements();
while (e.hasMoreElements())
 System.out.println(e.nextElement());

autoboxing happens
primitive values are converted into Integer objects

java.util.Vector

and its

We can create Enumeration object  
by using elements() method but  
elements() method returns an iterator  
so call v.elements()

(here Enumeration object means not its implementation class object  
↓ to get object E.g. class().getName())

### Limitations of Enumeration

- By using Enumeration we can perform only retrieval operation and we can't perform removal of Objects operation
- Enumeration is applicable only for legacy classes and hence it is not a universal cursor.
- To overcome the above limitations we should go for Iterator

## Iterator (I)

- We can use Iterator cursor to retrieve the elements one by one from any Collection object

- We can get Iterator object by using `Iterator()` method of Collection (I) Interface.

```
Public Iterator iterator();
```

Eg:

```
Iterator itr = c.iterator();
```

Once if we get Iterator object we can call the following methods on that object.

- Public boolean `hasNext()`;
- Public Object `next()`;
- Public void `remove()`;

Eg:

```
import java.util.*;
class IteratorDemo
{
    public static void main (String [] args)
    {
        ArrayList l = new ArrayList ();
        for (int i=0; i<10; i++)
        {
            l.add (i);
        }
        System.out.println (l); [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        Iterator itr = l.iterator ();
        while (itr.hasNext ())
        {
            Integer I = (Integer) itr.next ();
            if (I % 2 == 0)
                System.out.println (I); [0, 2, 4, 6, 8]
            else
                itr.remove ();
        }
    }
}
```

### Limitations of Iterator

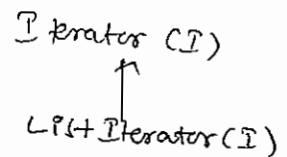
- By using Iterator we can perform read and remove operations only and we can't perform addition of new objects and replacement operation
  - By using Enumeration and Iterator we can read the elements only in forward direction and we can't read in backward direction
- To overcome the above limitations we should go for `ListIterator`.

## List Iterators

- We can use ListIterator to retrieve the elements one by one from Collection object.

- ListIterator is the child interface of Iterator

- we can get ListIterator object by using listIterator() method of List Interface



```
public ListIterator listIterator()
```

```
ListIterator ltr = l.listIterator();
```

- Once if we get ListIterator object we can call the following methods on that object

hasNext()  
next()  
nextIndex()

} forward direction

hasPrevious()  
previous()  
previousIndex()

} backward direction

remove()  
add (Object newObj)  
set (Object newObj)

Q36

import java.util;

class Test

```

{ public static void main(String[] args)
  {
    LinkedList l = new LinkedList();
    l.add("chiku");
    l.add("balaiyah");
    l.add("nag");
    l.add("avenger");
    System.out.println(l);
    ListIterator ltr = l.listIterator();
    while(ltr.hasNext())
    {
      String s = (String) ltr.next();
      if(s.equals("chiku"))
        ltr.set("cherries");
    }
  }
}
  
```

```

if (s.equals("charu"))
{
    lto.set("charan");
}

```

```

if (s.equals("nag"))
{
    lto.add("charu");
}

```

```

if (s.equals("renky"))
{
    lto.remove();
}

```

## Limitations of ListIterator

- Even though ListIterator is more powerful cursor but it's main limitation is it is applicable only for List objects and it is not a universal cursor.

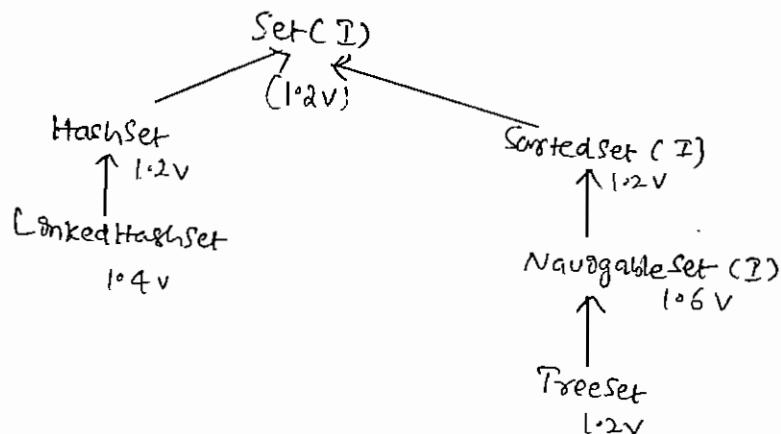
11-08-2014

## Comparison of Enumeration, Iterator and ListIterator

Property	Enumeration	Iterator	ListIterator
① How to get?	By using elements() method of Vector class	By using iterator() method of Collection Interface	By using listIterator() method of List interface
② Applicable for	only legacy classes	any Collection object	any List object
③ Access permissions	only read	read and remove	read, remove, add and replacement
④ Cursor movement	only in forward direction (unidirectional)	only in forward direction (unidirectional)	Both in forward and backward direction (Bi-directional)
⑤ methods	hasMoreElements(), next()	hasNext(), next() and remove()	hasNext(), next(), remove, nextIndex(), hasPrevious(), previous, previousIndex(), add(-), set(-), remove
⑥ Is it legacy?	Yes	No	No

## Set (I)

- It is the child interface of Collection
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order won't be preserved then we should go for Set(I).
- Set(I) interface doesn't contain any methods and hence we will use Collection interface methods only.



## HashSet

- 1) The underlined data structure is HashTable.
- 2) duplicates are not allowed
- 3) insertion order won't be preserved and it is based on HashCode of the objects
- 4) Heterogeneous Objects are allowed
- 5) Null insertion is possible (only once) ( $\because$  because duplicates are not allowed)

## Constructors

1) `HashSet h = new HashSet();`

→ creates an empty HashSet object with default initial capacity 16 and default fill ratio 0.75

→ fill ratio is also known as load-factor

(fill ratio  $\rightarrow$  after an  $O(1)$  insertion;  $O(n)$  search)

2) `HashSet h = new HashSet(int initialCapacity);`

creates an empty HashSet object with the specified initial capacity and default fill ratio 0.75

- 3) `HashSet h = new HashSet(int initialCapacity, float fillRatio);`
- 4) `HashSet h = new HashSet(Collection c)`

→ HashSet won't allow duplicates if we are trying to add a duplicate we won't get any compiletime error (or) runtime exception, simply add() method returns 'false'

Ex:

```
import java.util.*;
class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet h = new HashSet();
        h.add("B");
        h.add("c");
        h.add("D");
        h.add("Z");
        h.add(null);
        h.add(10);
        System.out.println(h.add("z")); // false
        System.out.println(h); [null, D, B, c, 10, Z]
```

Add() , if value  
has been  
added  
true  
false  
null  
initialCapacity  
fillRatio  
100% full

↳ here off by one error occurs  
because insertion order not preserved.

Hashcode(),  
equals()

- HashSet implements both Serializable and Cloneable interfaces but not RandomAccess
- HashSet is best suitable if our frequent requirement is searching operation.

## LinkedHashSet

- ↳ It is exactly same as HashSet except the following differences

<u>HashSet</u>	<u>LinkedHashSet</u>
① The underlined data structure is HashTable	① The underlined datastructure is combination of LinkedList and HashTable
② Insertion order won't be preserved	② Insertion order will be preserved
③ Introduced in 1.2v	③ Introduced in 1.4v
The important application areas of LinkedHashSet & LinkedHashMap are to implement Cache based applications where duplicates are not allowed and insertion order should be preserved.	

- In the above program If we replace HashSet with LinkedHashSet then the output is

$O(P \rightarrow [B, C, D, 2, null, 10])$

## SortedSet(I)

- It is the child interface of Set(I)
- If we want to represent a group of individual objects as a single entity without duplicates and according to some sorting order then we should go for SortedSet(I)

methods

Object first();

returns first element of the SortedSet

Object last();

returns last element of the SortedSet

SortedSet headSet(Object obj)

returns SortedSet whose elements are  $< obj$

SortedSet tailSet(Object obj)

returns SortedSet whose elements are  $\geq obj$

SortedSet subset(Object obj1, Object obj2)

returns SortedSet whose elements are  $\geq obj_1$  and  $< obj_2$

Comparator comparator()

returns Comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null.

Ex:

$[100, 200, 300, 400, 500, 600, 700]$

first()  $\rightarrow 100$

last()  $\rightarrow 700$

headSet(400)  $\rightarrow [100, 200, 300]$

tailSet(400)  $\rightarrow [400, 500, 600, 700]$

subset(200, 600)  $\rightarrow [200, 300, 400, 500]$

comparator()  $\rightarrow null$

$\hookrightarrow$  we are using default sorting order these

## TreeSet

- 1) The underlying data structure is Balanced Tree
- 2) duplicate objects are not allowed
- 3) insertion order won't be preserved and it is according to some sort of order
- 4) Heterogeneous objects are not allowed. by mistake if we are trying to add heterogeneous object then we will get ClassCastException
- 5) Null insertion is possible (only once) [null not allowed]

### Constructors

- ① TreeSet t = new TreeSet();  
Creates an empty TreeSet object where objects will be inserted according to Default natural sorting order (Dnso)
- ② TreeSet t = new TreeSet(Comparator c);  
Creates an empty TreeSet object where objects will be inserted according to customized sorting order (CSO) specified by Comparator
- ③ TreeSet t = new TreeSet(SortedSet s);  
Public TreeSet(SortedSet s);
- ④ TreeSet t = new TreeSet(Collection c);  
Public TreeSet(Collection c);

Note: for a number type objects ascending order. where as default natural sorting order is for strings it is alphabetical order

Eg:  
import java.util.\*;  
class TreeSetDemo

```

public static void main (String [] args)
{
    TreeSet t = new TreeSet();
    t.add ("A");
    t.add ("Z");
    t.add ("L");
    t.add ("B");
    t.add ("A");
    //t.add (10); R.E → ClassCastException
    //t.add (non); R.G → NullPointerException
    System.out.println(t);
    System.out.println("SOP(t); → [ A, B, L, Z ].");
}
  
```

## null acceptance

- 1) for the empty TreeSet has the first element we can insert null, but after inserting that null we can't insert any other element otherwise we will get ~~empty~~ runtime exception saying NullPointerException)
- 2) for the non empty TreeSet we can't insert null otherwise we will get ~~null~~ NullPointerException.

Note: from 1.7 version onwards ~~to~~ null insertion is not possible even as first element also in the empty TreeSet otherwise we will get NullPointerException.

Class test 1

```
Ex: 1) public class main(String[] args) {
    TreeSet t = new TreeSet();
    t.add(10);
    t.add(0);
    t.add(5);
    t.add(20);
    t.add(15);
    t.add(20);
}
```

O/P → NullPointerException

Ex: 2)

Class test 2

```
public class main(String[] args) {
    TreeSet t = new TreeSet();
    t.add(new StringBuffer("Rosa"));
    t.add(new StringBuffer("Namananend"));
    t.add(new StringBuffer("Shobhaani"));
    t.add(new StringBuffer("Ramulamma"));
    t.add(new StringBuffer("Sharmila"));
}
```

O/P → ClassCastException.

are

- \* whenever we depending on Default natural sorting order (DNSO) then objects should be compulsorily Homogeneous & Comparable.
- \* An object is set to be Comparable iff the corresponding class implements Comparable(I) interface.
- \* Comparable(I) interface present in java.lang package and defines only one method,

```
public int CompareTo(Object obj)
```

→ JVM internally uses this `CompareTo()` method to perform default natural sorting order in the following way.

`Obj1.compareTo(Obj2)`

(Compare with inner)

- returns -ve value ift obj<sub>1</sub> has to come before obj<sub>2</sub>
- returns +ve value ift obj<sub>1</sub> has to come after obj<sub>2</sub>
- returns 0 ift both obj<sub>1</sub> and obj<sub>2</sub> are same

Ex: `import java.util.*`  
`Class Test`

Comparing

```
    public static void main(String[] args) {  
        System.out.println("A".compareTo("Z")); // -25  
        System.out.println("Z".compareTo("A")); // 25  
        System.out.println("A".compareTo("A")); // 0
```

Internal process logic (BLL)

```
Treeset t = new TreeSet();
```

```
t.add("A");
```

[A]

```
t.add("Z"); → "Z".compareTo("A"); [A > Z]  
t.add("L"); → "L".compareTo("A"); { ← }  
                "L".compareTo("Z"); } ⇒ [A, L, Z]
```

```
t.add("B");
```

```
    → "B".compareTo("A"); } ⇒ [A, B, L, Z]  
    → "B".compareTo("L"); } ⇒ [A, B, L, Z]
```

```
t.add("A");
```

```
→ "A".compareTo("A"); } ⇒ [A, B, L, Z]
```

```
Treeset t = new TreeSet();
```

```
t.add("A");
```

```
t.add("Z");
```



JVM call `compareTo()` may add naturally  
may be before or after insertion.

```
t.add("L");
```

```
t.add("B");
```

```
t.add("A");
```

```
sop(t);
```

"Z".compareTo([A])

↓

JavaP 7 onwards sorting

• Priority

• HashSet, TreeSet

• Comparable

Sort & returning objects  
sequentially one by one

True object can be sorted  
Comparable interface

```
TreeSet t = new TreeSet();
```

```
t.add(10);
```

[10]

```
t.add(0); → 0.compareTo(10); } ⇒ [0, 10]
```

```
t.add(5); → +ve 5.compareTo(0); } ⇒ [0, 5, 10]
```

```
t.add(20); → -ve 5.compareTo(10); }
```

```
→ +ve 20.compareTo(0); } ⇒ [0, 5, 10, 20]
```

```
→ +ve 20.compareTo(5); } ⇒ [0, 5, 10, 20]
```

```
→ +ve 20.compareTo(10); }
```

```
t.add(15);
```



```
15.compareTo(0);
```

```
15.compareTo(5);
```

```
15.compareTo(10);
```

```
15.compareTo(20);
```

} ⇒ [0, 5, 10, 15, 20]

```
t.add(20);
```



```
20.compareTo(0);
```

```
20.compareTo(5);
```

```
20.compareTo(10);
```

```
20.compareTo(15);
```

```
20.compareTo(20);
```

} ⇒ [0, 5, 10, 15, 20]

\*\*

① String class and all wrapper classes already implemented Comparable interface hence these objects are Comparable objects but StringBuffer class doesn't implement Comparable (I) interface. Hence

• Hence StringBuffer object are non Comparable objects

② If we are trying to apply default natural sorting order for non Comparable objects then we will get runtime exception saying ClassCastException.

③ if we are not satisfying with Default natural sorting order (0 to 9) if default natural sorting order is not available then to define our own sorting we should go for Comparator(I) Interface

## Comparator (I)

- It is an interface present in `java.util` package
- It defines 2 methods
  - ① `Compare`
  - ② `equals`

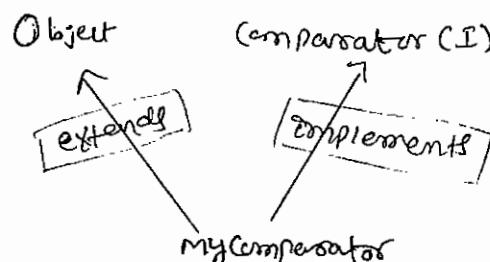
## Comparator (II)

→ `Public int Compare (Object obj1, Object obj2)`

- It returns -ve value iff `obj1` has to come before `obj2`.
- It returns +ve value iff `obj1` has to come after `obj2`.
- returning 0 If both `obj1` and `obj2` are same.

→ `Public boolean equals (Object o)`

- Whenever we are implementing `Comparator (I)` interface it is mandatory to provide implementation for `Compare ()` method and it is optional to implement `equals ()` method, because through `Object` class `equals ()` method implementation is available in our class.



\* `MyComparator (Implementation)`  
`Object` is a super class  
super class methods are  
extended in sub class

- Write a program to insert integer objects into `Treeset` where the sorting order is descending order.

```

import java.util.*;
class TreeSetDemo3
{
    public static void main (String [] args)
    {
        TreeSet t = new TreeSet (new MyComparator ());
        t.add (10);
        t.add (5);
        t.add (0);
        t.add (20);
        t.add (15);
        t.add (20);
        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 100; // any +ve value will
                        // return value is +ve
        else if (i1 > i2)
            return -100; // if obj1 and obj2 are swapped
                        // then obj2
        else return 0;          [20, 15, 10, 5, 0]
    }
}

```

Various possible implementations of `Compare()` method.

```

class MyComparator implements Comparator
{
    public int compare (Object obj1, Object obj2)
    {
        Integer i1 = (Integer) obj1;
        Integer i2 = (Integer) obj2;
        if return i1.compareTo(i2); → [0, 5, 10, 15, 20] Ascending order
        if return i2.compareTo(i1); → [20, 15, 10, 5, 0] Descending order
        if return -i1.compareTo(i2); → [20, 15, 10, 5, 0] Descending order
        if return -i2.compareTo(i1); → [0, 5, 10, 15, 20] Ascending order
        if return -100; → [20, 15, 10, 5, 0] Reverse order or Insertion
                        (If return value is -ve always object come before, what ever may be the order)
        if return +100; → [0, 10, 5, 20, 15] Insertion order
        return 0; → [10]
                    (only first element comes as obj, second element onwards it take any
                     element as duplicate)
    }
}

```

→ whenever we are defining customized sorting order (CSO) by implementing `Comparator(I)` interface then internally JVM uses `Compare()` method to perform sorting order

- (Q) Write a program to insert String objects into TreeSet where the sorting order should be reverse order of alphabetical order

```

import java.util.*;
class TreeSetDemo4
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("A");
        t.add("Z");
        t.add("L");
        t.add("B");
        t.add("A");
        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); // reverse order
        return -s1.compareTo(s2); // OP[ Z, L, B, A]
    }
}

```

- (Q) Write a program to insert StringBuffer objects into TreeSet where the sorting order should be alphabetical order.

```

import java.util.*;
class TreeSetDemo5
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add(new StringBuffer("Raja"));
        t.add(new StringBuffer("Nannapaneni"));
        t.add(new StringBuffer("ShobhaRani"));
        t.add(new StringBuffer("Ramulamma"));
        t.add(new StringBuffer("Sharmila"));
        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); // OP → Nannapaneni, Ramulamma, Raja, Sharmila, ShobhaRani
    }
}

```

(P) Write a program to insert String and StringBuffer objects into TreeSet where the sorting order should be increasing length order. If any two objects are having equal length then alphabetical order should be considered.

```

import java.util.*;
class TreeSetDemo6
{
    public static void main (String [] args)
    {
        TreeSet t = new TreeSet (new MyComparator ());
        t.add ("A");
        t.add ("AA");
        t.add (new StringBuffer ("ABC"));
        t.add (new StringBuffer ("ABCD"));
        t.add ("XX");
        t.add ("Z");
        System.out.println (t);
    }
}

class MyComparator implements Comparator
{
    public int compare (Object obj1, Object obj2)
    {
        String s1 = obj1.toString ();
        String s2 = obj2.toString ();
        int l1 = s1.length ();
        int l2 = s2.length ();
        if (l1 < l2)
            return -1;
        else if (l1 > l2)
            return +1;
        else
            return s1.compareTo (s2);
    }
}

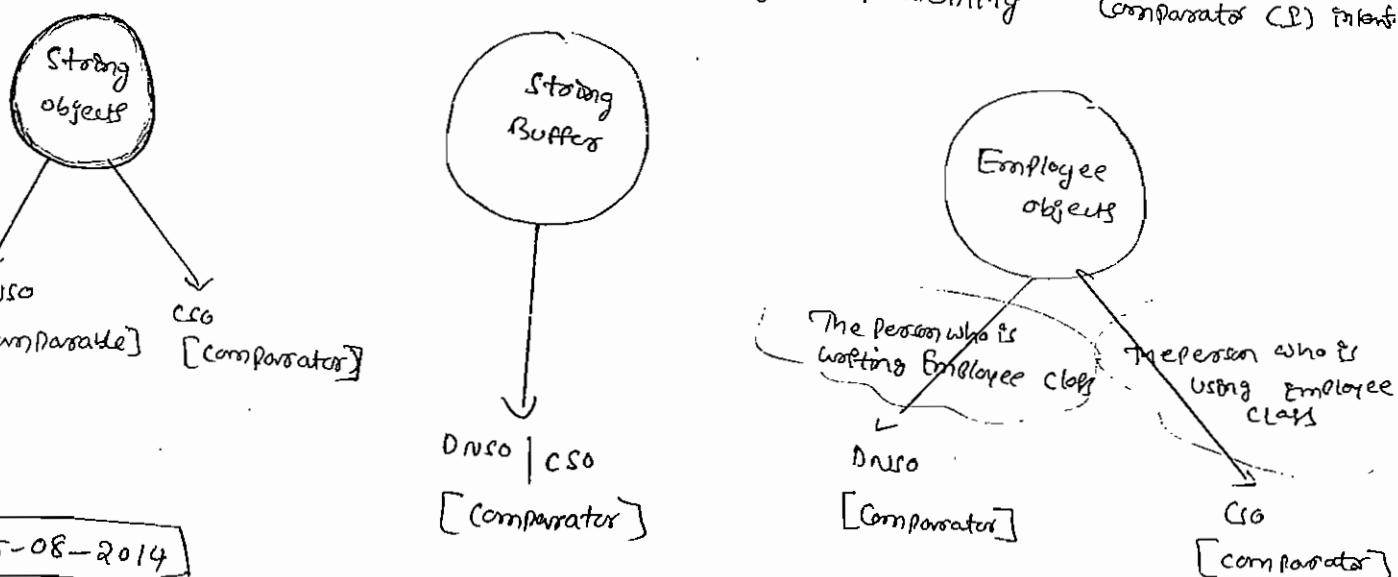
```

Output → [A, Z, AA, XX, ABC, ABCD]

- If we are defining our own sorting order by implementing Comparator interface then objects need not be homogeneous and need not be comparable.  
 i.e. Objects can be heterogeneous and non Comparable objects also.

## Comparable vs Comparator

- ① for predefined Comparable objects like String objects default natural sorting order (DNSO) is already available and if we are not satisfying with default natural sorting order then we can define our own sorting order by implementing Comparator interface.
- ② for the predefined non Comparable objects like String Buffer objects default natural sorting order is not already available and hence to define our own order we have to use Comparator object.
- ③ for the userdefined objects like Employee objects the person whom ever developing Employee class is responsible to define default natural sorting order by implementing Comparable interface.  
The person whom ever using Employee objects is not satisfied with default natural sorting order then that person can provide customized sorting order (CSO) by implementing Comparator (C) interface.



15-08-2014

Ex: import java.util.\*;

```

class Employee implements Comparable {
    int id;
    String name;
    Employee(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
    public String toString()
    {
        return id + "..." + name;
    }
    public int compareTo(Object obj)
    {
        int id1 = ((Employee)obj).id;
        Employee e = (Employee)obj;
    }
}
  
```

```

int id1 = e.getId();
if (id1 < id2)
    return -1;
else if (id1 > id2)
    return +1;
else return 0;
}

class CompDemo
{
    public static void main (String[] args)
    {
        Employee e1 = new Employee (100, "Chou");
        Employee e2 = new Employee (200, "Salaiyah");
        Employee e3 = new Employee (300, "Nag");
        Employee e4 = new Employee (500, "Venky");
        Employee e5 = new Employee (100, "Chou");
        TreeSet t1 = new TreeSet ();
        t1.add (e1);
        t1.add (e2);
        t1.add (e3);
        t1.add (e4);
        t1.add (e5);
        System.out.println (t1);
        TreeSet t2 = new TreeSet (new MyComparator ());
        t2.add (e1);
        t2.add (e2);
        t2.add (e3);
        t2.add (e4);
        t2.add (e5);
        System.out.println (t2);
        TreeSet t3 = new TreeSet (new MyComparator (1));
        t3.add (e1);
        t3.add (e2);
        t3.add (e3);
        t3.add (e4);
        t3.add (e5);
        System.out.println (t3);
    }
}

```

```

class MyComparator implements Comparator
{
    public int compare (Object obj1, Object obj2)
    {
        Employee e1 = (Employee) obj1;
        Employee e2 = (Employee) obj2;
        String s1 = e1.getName();
        String s2 = e2.getName();
        if (s1 < s2)
            return -1;
        else if (s1 > s2)
            return +1;
        else return 0;
    }
}

```

Q19

[100---chis, 300---nag, 500---venky, 700---balaiah]

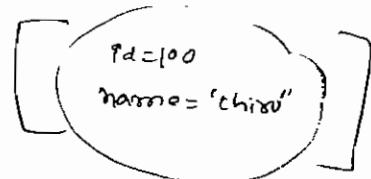
(19)

[500---venky, 300---nag, 100---chis, 700---balaiah]

```
TreeSet t1 = new TreeSet();
t1.add(e1);
t2.add(e2);
```

~~t1~~

e2.compareTo(e1);



id=700  
name = " balaiah". compareTo

id<sub>1</sub>=700 & id<sub>2</sub>=100

)

id=100  
name = "chis"

id=700  
name = " balaiah"

differences b/w Comparable & Comparator

#### Comparable (I)

- ① It is meant for default natural sorting order (DNO)
- ② Objects should be both homogeneous and comparable.
- ③ Present in java.lang Package
- ④ It defines only one method `compareTo()`
- ⑤ String class and all wrapper classes already implemented Comparable (I) interface.

#### Comparator (II)

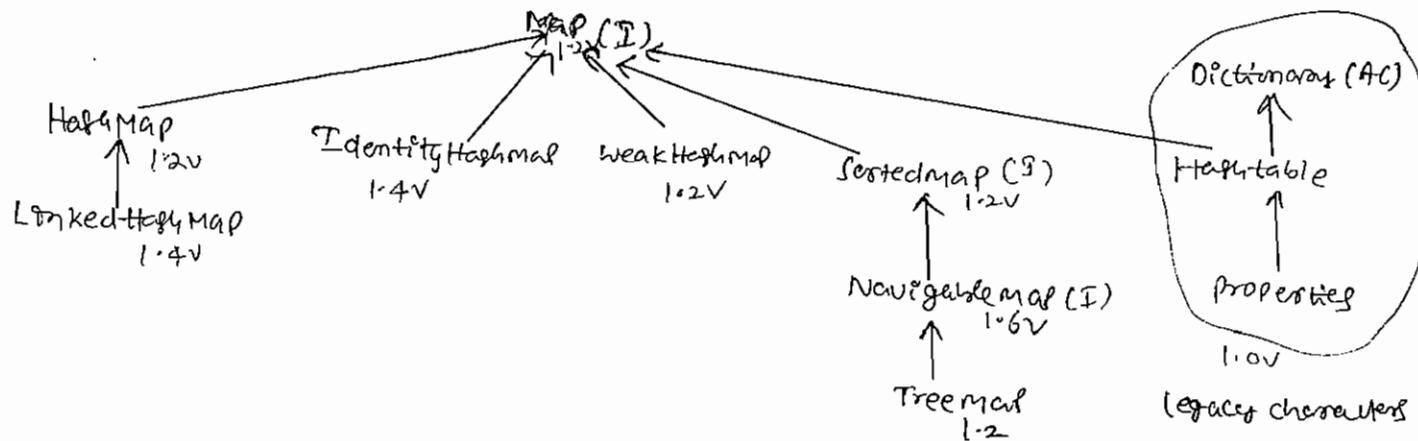
- ① It is meant for customized sorting order
- ② Objects need not be homogeneous and comparable
- ③ Present in java.util package
- ④ It defines 2 methods `compare()` & `equals()`
- ⑤ The only implementation classes which are implementing Comparator (II) interface are Collator & Rule Based Collator

## Comparison between Set Implementation Classes

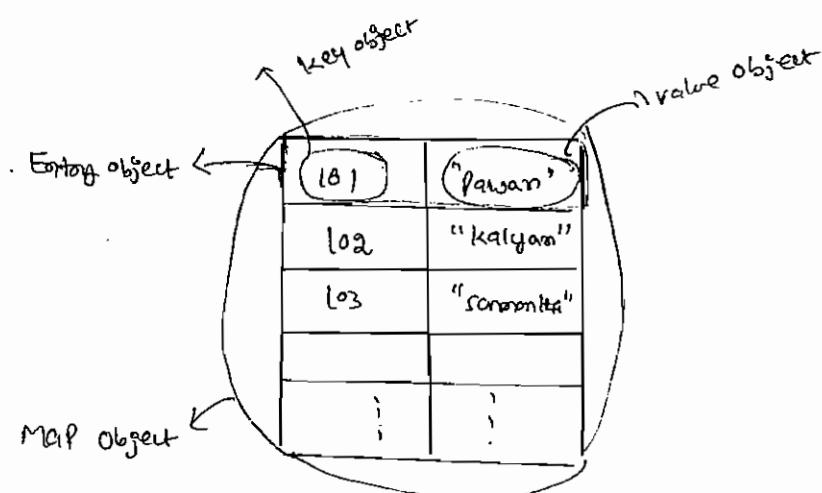
Property	HashSet	LinkedHashSet	TreeSet
① Underlying data structure	Hashtable	LinkedList + Hashtable	Balanced Tree
② Duplicate Objects	Not allowed	Not allowed	Not allowed
③ Insertion order	Not preserved	Preserved	Not preserved
④ Sorting order	Not available	Not available	Available
⑤ Heterogeneous Objects	Allowed	Allowed	Not allowed Otherwise we will get ClassCastException
⑥ Null insertion	Possible (only once)	Possible (only once)	for empty TreeSet as first element null insertion is possible but after inserting it null we can't insert any other element otherwise we will get <u>NullPointerException</u>
⑦ Introduced in	1.2v	1.4v	1.2v → for the non empty TreeSet null insertion is not possible otherwise we will get <u>NullPointerException</u>

## Map (Interface)

- 1) If we want to represent a group of objects as key, value pairs  
then we should go for Map interface.
- 2) Map is not child interface of Collection



- In Map both key and value are objects and each key, value pair is called as an entry i.e. A map object represents group of entries



### Map Interface Methods

- ① Object put(Object key, Object value)

To insert a key-value pair into Map object

If the specified key is already available then old value will be replaced with new value and returns old value

- ② ~~Object~~ void putAll(Map m)
- ③ Object get(Object key)
- ④ Object remove(Object key)
- ⑤ boolean containsKey(Object key)
- ⑥ boolean containsValue(Object value)
- ⑦ boolean isEmpty()
- ⑧ int size()
- ⑨ void clear();
- ⑩ Set keySet() → only keys we get
- ⑪ Collection values() → Collection view of Map object
- ⑫ Set entrySet() → only values we get.

### Entry Interface

- 1) In Map object each key, value pair is called an Entry
- 2) With out existing Map object there is no chance of existing Entry object.  
Hence Sun people declared entry interface inside Map interface.  
(Strong association which is also known as Composition)

interface Map

{

====

Interface Entry

{

Object getKey();

// method

Object getValue();

Object setValue(Object newValue);

y } A to replace the value.

y }

## HashMap (18-08-14)

- 1) The underlying datastructure is hashtable
- 2) duplicate keys are not allowed but values can be duplicated
- 3) insertion order won't be preserved and it is based on hashCode of the keys
- 4) heterogeneous objects are allowed for both key and values
- 5) null insertion is possible for both key (only once) and values (any number of times)

### Constructors

① HashMap h = new HashMap();

Creates an empty HashMap object with default initial capacity 16 and default fill ratio 0.75

② HashMap h = new HashMap(int initialCapacity);

③ HashMap h = new HashMap(int initialCapacity, float fillRatio);

④ HashMap h = new HashMap(Map m);

↳ If any Map class here.

import java.util.\*;

class HashmapDemo

{ public static void main(String[] args)

HashMap h = new HashMap();

h.put("chou", 700);

h.put("venky", 900);

h.put("balaiyah", 200);

h.put("nag", 500);

{  
    Put -> old values replaced  
                  keys  
    with new values  
    duplicates not allowed

```

SOP(h);           → entries are not synchronized
SOP(h.put("chiru", 1000));

Set s = h.keySet();
SOP(s);
Collection c = h.values();
SOP(c);
Set s1 = h.entrySet(); // return key, values pair
SOP(s1);

Iterator itr = s1.iterator();
while (itr.hasNext()) {           ← entry present in map
    Map.Entry e = (Map.Entry) itr.next();
    SOP(e.getKey() + "----" + e.getValue());   ← for printing set values
    if (e.getKey().equals("mag"))
        e.setValue(10000);
}
SOP(h);
}

```

## Differences b/w HashMap and Hashtable

HashMap	Hashtable
1) No method is synchronized in HashMap	1) Every method present inside Hashtable is by default synchronized
2) multiple threads are allowed to operate on HashMap object simultaneously and hence it is not thread safe.	2) only one thread is allowed to open Hashtable object at a time and hence it is thread safe
3) Threads are not required to wait so that relatively performance is high	3) Threads are required to wait so that relatively performance is low
4) introduced in 1.0 version and hence it is not legacy	4) introduced in 1.0 version and hence it is legacy.
*5) null insertion is possible for both key (only once) and values (any number of times)	*5) null insertion is not possible for both key and values even in the first entry also otherwise we will get <u>NullPointerException</u>

## Synchronized version of HashMap

① How to get Synchronized version of HashMap  
By default HashMap is non synchronized but we can get synchronized version of HashMap by using `SynchronizedMap()` method of Collections class.

```
Public static Map SynchronizedMap(Map m) {
```

Ex:

```
HashMap m = new HashMap();
Map m1 = Collections.SynchronizedMap(m);
```

→ Similarly we can get synchronized versions of List and Set by using the following methods of Collections class

① Public static List `SynchronizedList(List l)`

② Public static Set `SynchronizedSet(Set s)`

## LinkedHashMap

It is exactly same as HashMap except the following differences

HashMap	LinkedHashMap
① The underlying data structure for HashMap is hashtable	① The underlying data structure for LinkedHashMap is combination of LinkedList and hashtable
② Insertion order won't be preserved and it is based on Hash code of the keys	② insertion order will be preserved
③ Introduced in 1.2v	③ Introduced in 1.4v

Ex: `HashMap m = new HashMap();`

```
m.put("chinu", 700);
m.put("venky", 900);
m.put("balu", 200);
m.put("nag", 500);
```

`System.out.println(m);`

Output

{ chinu=700, bala=200, venky=900, nag=500 }

In the above code if we replace HashMap object creation with them the output is

{ chinu=700, venky=900, bala=200, nag=500 }

(In Java we use LinkedHashMap)

LinkedHashMap

①

## IdentityHashMap

It is exactly same as HashMap except the following difference.

- In case of HashMap to identify duplicate keys internally JVM uses • equals() method, which is meant for content comparison.
- whereas in case of IdentityHashMap to identify duplicate keys internally JVM uses == operator, which is meant for reference comparison.

Ex:

```
import java.util.*;
class Test {
    public static void main (String [] args) {
        HashMap m = new HashMap();
        Integer i1 = new Integer(10);
        Integer i2 = new Integer(10);
        m.put (i1, "Pawan");
        m.put (i2, "Kalyan");
        System.out.println(m);
    }
}
```

→ Since references  
made in != operator  
comparison

```
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
System.out.println(i1 == i2); → false
System.out.println(i1.equals(i2)); → true
```

i1  $\checkmark$  (10)

i2  $\checkmark$  (10)

Output → {10=Pawan, 10=Kalyan}

In the above example if we replace HashMap with IdentityHashMap then the output is {10=Pawan, 10=Kalyan}

## WeakHashMap

- It is exactly same as HashMap except the following difference.
- \* In case of HashMap if an object is not referred by any reference variable then also it is not eligible for garbage collection if that object is associated with HashMap.  
i.e. HashMap dominates garbage collector.

but in the case of WeakHashMap if an object is not referred by any reference variable then it is always eligible for GC (Garbage Collection) even though that object is associated with WeakHashMap

i.e. Garbage collector dominates WeakHashMap

Ex:

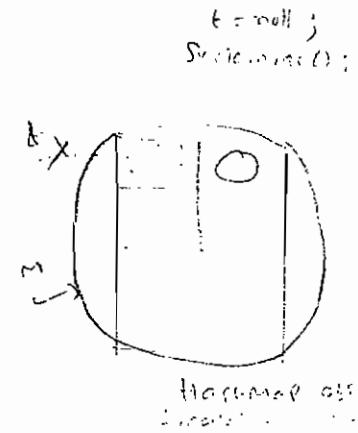
```
import java.util.*;
class WeakHashMapDemo
{
    public static void main(String[] args) throws
                                                Exception
    {
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t, "Srikanth");
        System.out.println(m);
        t=null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(m);
    }
}
```

```
class Temp
{
    public String toString()
    {
        return "temp";
    }
    public void finalize()
    {
        System.out.println("finalize called");
    }
}
```

O/P:  
{ temp = Srikanth }  
{ temp = Srikanth }

By the above example if we replace HashMap object with WeakHashMap  
then the O/P is

```
{ temp = Srikanth }
finalize called
{ }
```



## SortedMap (I)

If we want to represent a group of key,value pairs according to some sorting order of keys then we should go for SortedMap. It is the child interface of Map.

### Methods

- ① Object firstKey();
- ② Object lastKey();
- ③ SortedMap headMap(Object key)
- ④ SortedMap tailMap (Object key)
- ⑤ SortedMap subMap (Object key<sub>1</sub>, Object key<sub>2</sub>)
- ⑥ Comparator comparator()

## TreeMap

- ① The underlying datastructure for TreeMap is RedBlack Tree
- ② duplicate keys are not allowed but values can be duplicated.
- ③ insertion order won't be preserved and it is based on some sorting order of keys
- ④ Heterogeneous objects are not allowed for keys otherwise we will get ClassCastException. but for value heterogeneous objects are allowed
- ⑤ for empty TreeMap in the first entry null key is allowed but after inserting that entry we can't insert any other entry otherwise we will get NullPointerException  
for the non empty TreeMap we can't insert any entry with null key otherwise we will get NullPointerException.

Note: from 1.7 version onwards for empty TreeMap also null key is not allowed otherwise we will get NullPointerException

### Constructors

- ① TreeMap t = new TreeMap();  
 → key, value pairs will be inserted into TreeMap according default natural sorting order of keys.  
 → in this case keys should be both homogeneous and Comparable otherwise we will get ClassCastException

(2) TreeMap t = new TreeMap(Comparator c);  
 key, value pairs inserted into TreeMap according to customized  
 sorting order specified by Comparator object  
 In this case keys need not be Comparable. i.e. they can be objects also.  
 homogeneous and need not be heterogeneous and noncomparable

(3) TreeMap t = new TreeMap(SortedMap m);

(4) TreeMap t = new TreeMap(Map m);

Exe

```
import java.util.*;
class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap t = new TreeMap();
        t.put(100, "AAA");
        t.put(500, "0.5");
        t.put(200, "CCC");
        t.put(400, "LLL");
        t.put(300, 123);
        // t.put("sof", "XXX"); → R.O.C ClassCastException
        // t.put(null, "XXX"); → R.O.C NullPointerException
        System.out.println(t);
    }
}
```

O/P: { 100 = AAA, 200 = CCC, 300 = 123, 400 = LLL, 500 = 0.5 }

Ex 28

```
import java.util.*;
class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap t = new TreeMap(new MyComparator());
        t.put("A", "AAA");
        t.put("Z", "0.5");
        t.put("L", "CCC");
        t.put("B", "LLL");
        t.put("A", 123);
        System.out.println(t);
    }
}
```

1. We create  
 MyComparator  
 class in the file  
 MyComparator.java  
 Our class  
 implements  
 Comparator  
 interface  
 implements  
 Comparable  
 interface  
 and  
 overrides  
 compare  
 method

class MyComparator implements Comparator  
 {  
 public int compare (Object obj1, Object obj2)  
 {  
 String s1 = (String) obj1;  
 String s2 = obj2.toString();  
 if return s2.compareTo(s1);  
 }  
 }  
 O/P: { Z=10.5, L=ccc, B=LLL, A=123 }

Ex 3)

```

import java.util.*;
class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap t = new TreeMap(new MyComparator());
        t.put(new StringBuffer("chiru"), 100);
        t.put(new StringBuffer("balalak"), 500);
        t.put(new StringBuffer("nag"), "sri");
        t.put(new StringBuffer("venky"), "komit");
        t.put(135, 10.5);
        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);
    }

```

$$\text{O/P}_e \quad \left\{ 135 = 10.5, \text{ balansah} = 500, \text{ chisau} = 100, \text{ mag} = 50^{\circ}, \text{ Venky} = 100 \right.$$

## Hashtable

- ① The underlying datastructure for Hashtable is Hashtable
- ② Duplicate keys are not allowed but values can be duplicated
- ③ Insertion order won't be preserved and it is based on hashCode of the keys
- ④ Heterogeneous Objects are allowed for both key and values
- ⑤ Null insertion is not possible for both key and value even in the first entry also otherwise we will get NullPointerException
- ⑥ Every method present inside Hashtable is by default synchronized hence only one thread is allowed to operate Hashtable object at a time and hence it is thread safe.

## Constructors

1) Hashtable h = new Hashtable();

Creates an empty hashtable object with default initial capacity 11 (eleven) and default fill ratio 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);

## Ex:

```
import java.util.*;
class HashtableDemo
{
    public static void main(String[] args)
    {
        Hashtable h = new Hashtable();
        // h.put("A", null); → R.E. of NullPointerException
        h.put("A", 100);
        h.put(123, "Srikant");
        h.put("Z", 157);
        h.put(new StringBuffer("mn"), 147);
        h.put("Z", 200);
        System.out.println(h);
    }
}
```

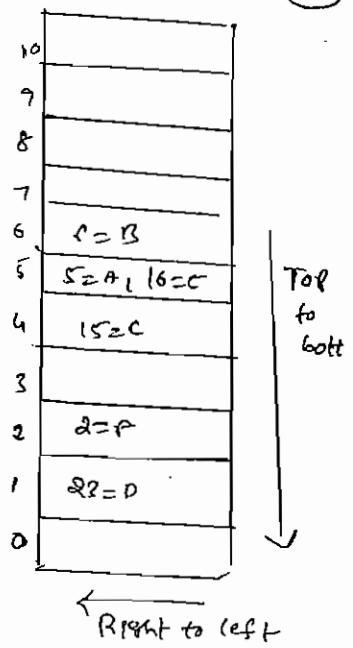
O/P { A=100, mn=147, Z=200, 123=Srikant }

Example

```

import java.util.*;
class HashtableDemo
{
    public static void main(String[] args)
    {
        Hashtable h = new Hashtable();
        h.put(new Temp(5), "A");
        h.put(new Temp(6), "B");
        h.put(new Temp(15), "C");
        h.put(new Temp(23), "D");
        h.put(new Temp(16), "E");
        h.put(new Temp(2), "F");
        // h.put("durga", null); line
        System.out.println(h);
    }
}

```



## Class Temp

```

{
    int i;
    Temp(int i)
    {
        this.i = i;
    }
    public String toString()
    {
        return i + " ";
    }
    public int hashCode()
    {
        return i;
    }
}

```

O/P: {6=B, 16=E, 5=A, 15=C, 2=F, 23=D}

Properties

- In our Java program if anything changes more frequently then it is not recommended to hardcore such kind of details in our program.
- because for every small change in the program we required to perform recompilation, rebuilding the application & reloading the application sometimes even server restart also required, which impacts clients.
- If we want to maintain the data which is changing frequently then we have to use a separate file called properties file.
- In properties file we have to maintain the data in the form of key, value pairs and based on our requirement we should read that data into Java program.
- If we want to perform any changes we will perform those changes in the properties file and to reflect these changes just reloading the

application is more than enough, which don't impacts client business.

- We can use properties object to hold the data in our java application which is coming from properties file.

### Constructors

Properties p = new Properties();

Creates an empty Properties object

### Methods

① String setProperty(String name, String value)

to set a new property

② (String) getProperty(String name);

to get value associated with the specified property

③ (Enumeration) PropertyNames()

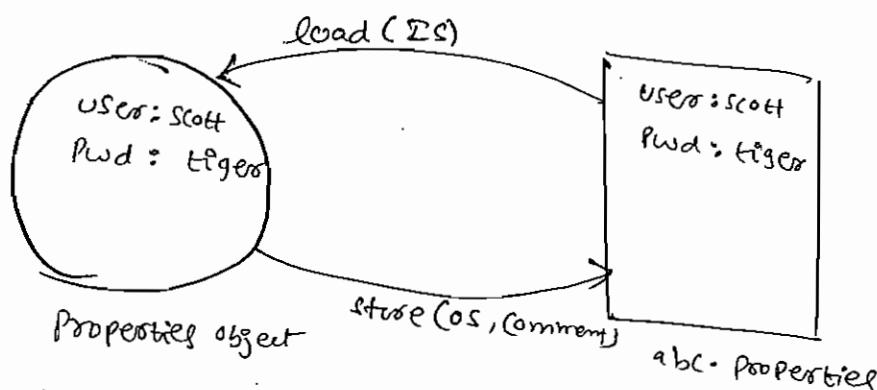
It returns all property names

④ public void load(InputStream is)

to load properties from properties file into java properties object.

⑤ public void store(OutputStream os, String comment)

Properties to store properties from java properties object into file



### ExB

```
import java.util.*;  
import java.io.*;  
class PropertiesDemo  
{  
    public static void main(String[] args) throws Exception  
    {  
        Properties p = new Properties();  
    }  
}
```

```

fileInputStream fis = new FileInputStream("abc.properties");
p.load(fis);
System.out.println(p);
String pwd = p.getProperty("pwd");
System.out.println(pwd);
p.setProperty("vernty", "ooo");
FileOutputStream fos = new FileOutputStream("abc.properties");
p.store(fos, "Updated by Sakant for core java collection---");
}
}

```

### abc.Properties (before executing the program)

User = Sakant  
pwd = Anushka

After executing the program

Updated by Sakant for core java collection  
vernty = ooo  
User = Sakant  
pwd = Anushka

### Ex 2

```

import java.util.*;
import java.io.*;
import java.sql.*;
class PropertiesDemo
{
    public static void main(String[] args) throws Exception
    {
        Properties p = new Properties();
        FileInputStream fis = new FileInputStream("dInfo.properties");
        p.load(fis);

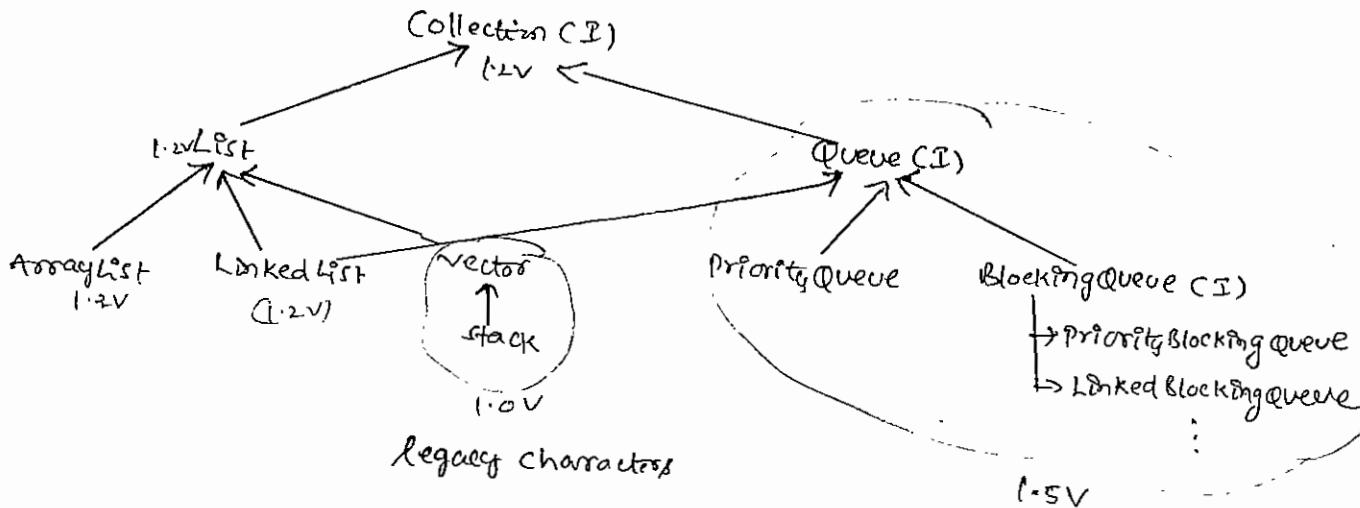
        String driver = p.getProperty("driver");
        String url = p.getProperty("url");
        String user = p.getProperty("user");
        String pwd = p.getProperty("pwd");
        Class.forName(driver);
        Connection con = DriverManager.getConnection(url, user, pwd);
    }
}

```

--

## Queue (I) [1.5V enhancement]

- ① It is the child interface of Collection
- ② If we want to represent a group of objects prior to processing then we should go for Queue (I) Interface.



- Usually Queue follows FirstInFirstOut (FIFO) order but based on our requirement we can provide our own ~~pre~~ priorities also (PriorityQueue)
- From 1.5V onwards LinkedList class implements Queue interface and Queue always follow FIFO order.

### Methods

- ① boolean offer(Object o)  
to add an object to onto the queue
- ② Object peek()  
to return head element of the queue. If queue is empty then this method returns null.
- ③ Object element()  
to return element of the queue. If queue is empty then this method raises RuntimeException
- ④ Object poll()  
to remove and return head element of the queue. If queue is empty then this method returns null.
- ⑤ Object remove()  
to remove and return head element of the queue. If queue is empty then this method raises RuntimeException

[ 22-08-14 ]

## Priority Queue

- It is a data structure which can be used to represent a group of objects prior to processing and according to some priority.
- Duplicate objects are not allowed.
- Insertion orders won't be preserved and it is according to some priority order.
- These priority orders can be either default natural sorting order or customized sorting order.
  - If it is default natural sorting order then Objects should be compulsory homogeneous and comparable otherwise we will get ClassCastException.
  - Null insertion is not possible even as first element also. otherwise we will get NullPointerException.

## Constructors

- ① Priority Queue q = new PriorityQueue();  
Creates an empty PriorityQueue object with default initial capacity 11 and objects will be inserted according to default priority i.e. default natural sorting order.
- ② Priority Queue q = new PriorityQueue(int initialCapacity);
- ③ Priority Queue q = new PriorityQueue(int initialCapacity, Comparator c);
- ④ Priority Queue q = new PriorityQueue(SortedSet S);
- ⑤ Priority Queue q = new PriorityQueue(Collection c);

## Ex:

```
import java.util.*;  
class PriorityQueueDemo  
{  
    public static void main(String[] args)  
    {  
        Priority Queue q = new PriorityQueue();  
        System.out.println(q.peek()); // null  
        System.out.println(q.element()); // R.E. NoSuchElementException  
        for(int i=0; i<10; i++)  
        {  
            q.offer(i);  
        }  
        System.out.println(q); // [0, 1/2, ... 9]  
        System.out.println(q.poll()); // 0  
        System.out.println(q.remove()); // 1  
        System.out.println(q); // [1/3, ..., 9] } } } } } }
```

→ Some of points:  
1/ P has offered 11

## Notes

Some operating systems may not provide proper support for PriorityQueues.

## Ex:

```

import java.util.*;
class PriorityQueueDemo2 {
    public static void main(String[] args) {
        PriorityQueue q = new PriorityQueue(15, new MyComparator());
        q.offer("A");
        q.offer("Z");
        q.offer("L");
        q.offer("B");
        System.out.println(q); // [Z, L, B, A]
    }
}

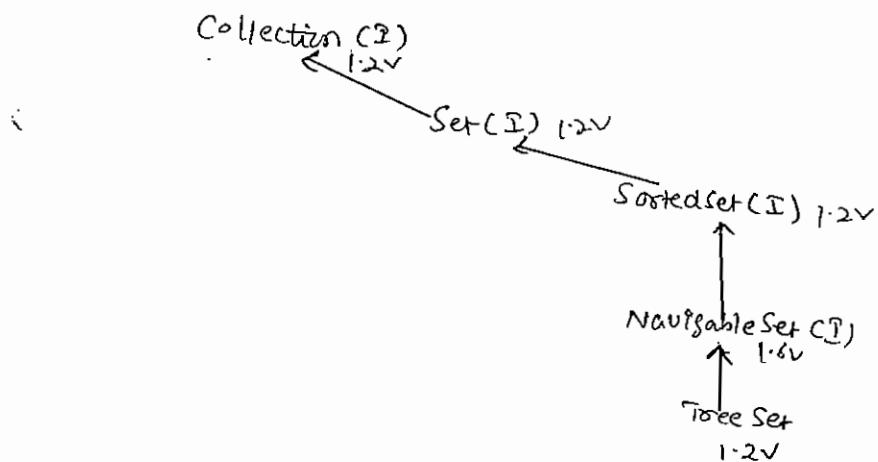
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = (String) obj1;
        String s2 = obj2.toString();
        return -s1.compareTo(s2);
    }
}

```

## 1.6v Enhancements

### (i) NavigableSet (I)

- It is the child interface of SortedSet
- It defines various methods for navigation purpose



methods

- ① `floor(e)`  
it returns highest element which is  $\leq e$
- ② `lower(e)`  
it returns highest element which is  $< e$
- ③ `ceiling(e)`  
it returns lowest element which is  $\geq e$
- ④ `higher(e)`  
it returns lowest element which is  $> e$
- ⑤ `PollFirst()`  
remove and return first element
- ⑥ `PollLast()`  
remove and return last element
- ⑦ `descendingSet()`  
it returns NavigableSet in reverse order

Example

```

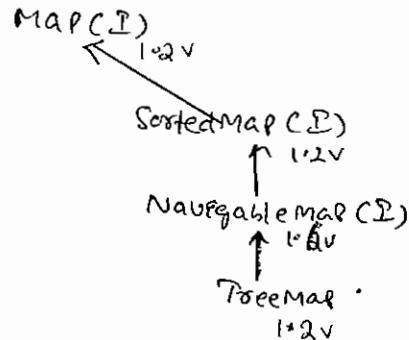
import java.util.*;
public static void main(String[] args) {
    TreeSet<Integer> t = new TreeSet<Integer>();
    t.add(100);
    t.add(200);
    t.add(300);
    t.add(400);
    t.add(500);
    t.add(600);
    t.add(700);
    System.out.println(t); // [100, 200, 300, 400, 500, 600, 700]
    System.out.println(t.ceiling(400)); // 400
    System.out.println(t.higher(400)); // 500
    System.out.println(t.floor(300)); // 300
    System.out.println(t.lower(300)); // 200
    System.out.println(t.PollFirst()); // 100
    System.out.println(t.PollLast()); // 700
    System.out.println(t.descendingSet()); // [600, 500, 400, 300, 200]
    System.out.println(t); // [200, 300, 400, 500, 600]
}
    
```

↑ returning reverse object

to provide more safety  
by allocating only integers

## (H) Navigable Map (I)

- It is the child interface of SortedMap
- It defines various methods for navigation purpose



### Methods

- ① floorKey (e)
- ② lowerKey (e)
- ③ ceilingKey (e)
- ④ higherKey (e)
- ⑤ PollFirstEntry ()
- ⑥ PollLastEntry ()
- ⑦ descendingMap ()

### Exe

```
import java.util.*;  
class NavigableMapDemo  
{  
    public static void main(String[] args)  
    {  
        TreeMap<String, String> t = new TreeMap<String, String>();  
        t.put("b", "banana");  
        t.put("c", "cat");  
        t.put("a", "apple");  
        t.put("d", "dog");  
        t.put("g", "gun");  
        System.out.println(a = apple, b = banana, c = cat, d = dog, g = gun);  
        System.out.println(t.ceilingKey("c"));  
        System.out.println(t.higherKey("e"));  
        System.out.println(t.floorKey("e"));  
        System.out.println(t.lowerKey("c"));  
        System.out.println(t.PollFirstEntry());  
        System.out.println(t.PollLastEntry());  
        System.out.println(t.descendingMap());  
        System.out.println(t);  
    }  
}
```

The code demonstrates the use of the `TreeMap` class and its methods. It inserts five key-value pairs: ("b", "banana"), ("c", "cat"), ("a", "apple"), ("d", "dog"), and ("g", "gun"). It then prints the values corresponding to the keys 'a' through 'g'. It uses the `ceilingKey`, `higherKey`, `floorKey`, and `lowerKey` methods to find the keys 'c', 'e', 'e', and 'c' respectively. It also uses `PollFirstEntry` and `PollLastEntry` to remove the first and last entries from the map. Finally, it prints the descending map, which lists the keys in reverse order: "d", "c", "b", "a", "g". The output is annotated with variable assignments: `a = apple`, `b = banana`, `c = cat`, `d = dog`, `g = gun`.

## Utility classes (Collections & Arrays) :-

### ① Collections

Collections class is an utility class present in `java.util` package to define several utility methods for Collection objects.

#### ① To Sort elements of List :

Collections class defines the following methods for this purpose

##### ① (Public static void sort(List l))

To sort based on default natural sorting order.

- In this case, compulsory List should contain only homogeneous & Comparable objects, otherwise we will get `RuntimeException` saying `ClassCastException`
- List should not contain null otherwise we will get `NullPointerException`

##### ② (Public static void sort(List l, Comparator c))

To sort based on customized sorting order.

Exe To sort elements of List according to natural sorting order :-  
 import `java.util.*;`

Class `CollectionDemo`

P r. v main(String[] args)

{ ArrayList l = new ArrayList();

l.add("Z");

l.add("A");

l.add("K");

l.add("N");

// l.add(new Integer(10)); → R.E. ClassCastException  
 // l.add(null); → R.E. NullPointerException

SOP("Before Sorting : "+l); O/P: [Z,A,K,N]

`Collections.sort(l);`

SOP("After Sorting : "+l); → O/P: [A,K,N,Z]

Ex-2 To sort elements of list according to customized sorting order:-

```
import java.util.*;  
class CollectionsSortDemo1  
{  
    public static void main(String[] args)  
    {  
        ArrayList l = new ArrayList();  
        l.add("Z");  
        l.add("A");  
        l.add("K");  
        l.add("L");  
        System.out.println("Before sorting : " + l); // O/P : [Z, A, K, L]  
        Collections.sort(l, new Comparator());  
        System.out.println("After sorting : " + l); // O/P : [Z, L, K, A]  
    }  
}
```

```
class MyComparator implements Comparator  
{  
    public int compare(Object obj1, Object obj2)  
    {  
        String s1 = (String) obj1;  
        String s2 = (String) obj2; // obj2.toString();  
        return s2.compareTo(s1);  
    }  
}
```

## (2) Searching elements of List:

① public static int binarySearch(List l, Object target)

If we ~~sort~~ are sorting list according to default natural sorting order then we have to use this ~~method~~ method.

② public static int binarySearch(List l, Object target, Comparator c)

If we sort list according to Comparator then we use this method.

Conclusion

- ① Internally the above search methods will use BinarySearch algorithm
- ② Before performing search operation compulsory List should be sorted otherwise we will get unpredictable results
- ③ Successful Search returns index
- ④ Unsuccessful Search returns insertion point.

- 5) Insertion point is the place where we can place target element in sorted list.
- 6) If the list is sorted according to Comparator then at the time of search operation also then we should pass the same Comparator object otherwise we will get unpredictable results.

Ex 1

List is sorted according to natural sorting order

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

```
class CollectionsSearchDemo
```

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

```
{
    ArrayList l = new ArrayList();
    l.add("2");
    l.add("4");
    l.add("M");
    l.add("K");
    l.add("A");
}
```

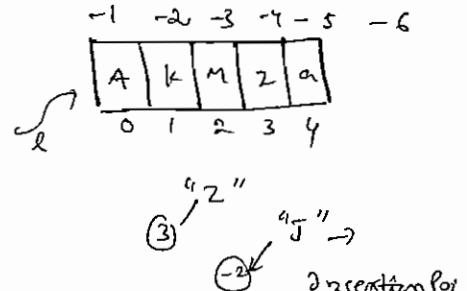
```
SOP(l); => O/P : [2, A, M, K, 4]
```

```
Collections.sort(l);
```

```
SOP(l); => O/P : [A, K, M, 2, 4]
```

```
SOP(Collections.binarySearch(l, "2")); => O/P : 3
```

```
SOP(Collections.binarySearch(l, "5")); => O/P : -2
```



(3)  $\nearrow$   $\nwarrow$   $\rightarrow$   
insertion point

Ex 2

List is sorted according to customized sorting order:-

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

```
class CollectionsSearchDemo1
```

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

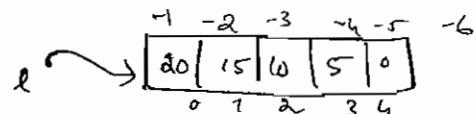
```
{
    ArrayList l = new ArrayList();
    l.add(15);
    l.add(0);
    l.add(20);
    l.add(10);
    l.add(5);
}
```

```
SOP(l); => O/P : [15, 0, 20, 10, 5]
```

```
Collections.sort(l, new MyComparator());
```

```
SOP(l); => O/P : [0, 5, 10, 15, 20]
```

```
SOP(Collections.binarySearch(l, 10, new MyComparator())); //2
```



```
SOP(Collections.binarySearch(lis, new MyComparator())); O/P: -3
```

```
SOP(Collections.binarySearch(lis, new MyComparator())); O/P: -6 (unpredictable)
```

3) Class MyComparator implements Comparator

```
public int compare(Object obj1, Object obj2){  
    Integer i1 = (Integer)obj1;  
    Integer i2 = (Integer)obj2;  
    return i2.compareTo(i1);}
```

Note: for the list of n elements

Range of successful search: 0 to n-1

Range of unsuccessful search: -(n+1) to -1

Total Result Range: -(n+1) to n-1

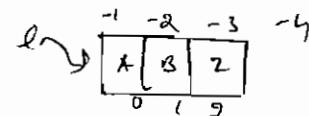
Ex: 3 elements

Range of successful search: 0 to 2

Range of unsuccessful search: -4 to -1

Total Result Range: -4 to 2

Reversing the elements of list:



```
Public static void reverse(List l):
```

Ex:

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

```
Class CollectingReverseDemo
```

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

```
    ArrayList l = new ArrayList();
```

```
    l.add(15);
```

```
    l.add(0);
```

```
    l.add(20);
```

```
    l.add(10);
```

```
    l.add(5);
```

```
SOP(l); // O/P: [15, 0, 20, 10, 5]
```

```
Collecting.reverse(l);
```

```
SOP(l); [5, 10, 20, 0, 15]
```

## reverse() vs reverseOrder()

- we can use `reverse()` method to reverse order of elements of `List`
  - we can use `reverseOrder()` method to get reversed `Comparator`
- Ex:

`Comparator c1 = Collection.reverseOrder(Comparator c)`



Descending order



Ascending order.

## (2) Arrays

`Arrays` class is an utility class to define several utility methods for `Array` objects.

### (1) Sorting elements of Array :-

`Public static void sort(primitive[] p)`

To sort according to default natural sorting order.

`Public static void sort(Object[] o)`

To sort according to Natural sorting order

`Public static void sort(Object[] o, Comparator c)`

To sort according to customized sorting order

Note: for `Object` type `Arrays` we can sort according to (NSO) natural sorting order (or) customized sorting order. But we can sort primitive array only based on (NSO) natural sorting order, but not based on customized sorting order (CSO).

### Ex:- To sort elements of Array :-

import `java.util.Arrays;`

import `java.util.Comparator;`

Class `ArraySortDemo`

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

```
    int[] a = {10, 5, 20, 11, 6};
```

```
    System.out.println("Primitive Arrays before sorting");
```

```
    for (int a1 : a)
```

```
        System.out.print(a1);    op: [10, 5, 20, 11, 6]
```

```
    Arrays.sort(a);
```

```

Sop (" primitive Array After sorting");
for (int a1: a)
{
    Sop (a1);    o/p: [5, 6, 10, 11, 20]
}
String [] s = {"A", "Z", "B"};
Sop (" Object Array before sorting");
for (String s1: s)
{
    Sop (s1);    o/p: [A, Z, B]
}
Arrays.sort (s);
Sop (" Object array After sorting");
for (String s1: s)
{
    Sop (s1);    o/p: [A, B, Z]
}
Arrays.sort (s, new MyComparator ());
Sop (" Object Array After sorting by comparator");
for (String s1: s)
{
    Sop (s1);    o/p: [Z, B, A]
}

```

```

class MyComparator implements Comparator
{
    public int compare (Object o1, Object o2)
    {
        String s1 = o1.toString ();
        String s2 = o2.toString ();
        return s2.compareTo (s1);
    }
}

```

## ② Searching elements of Array:

Arrays class defines the following methods.

1. **Public static int binarySearch (Primitive[] p, primitive target)**

If the primitive Array we have to use this sorted according to Natural sorting order then

2. public static int binarySearch (Object[] a, Object target)

If the Object Array sorted according to NSO then we have to use this method.

(3) public static int binarySearch (Object[] a, Object target,

If the Object Array sorted according to Comparator c)

we have to use this method.

Note: All rules of Arrays class binarySearch() method are same as Collections class binarySearch() method.

To search elements of Arrays -

import java.util.\*;

class ArraySearchDemo

{ public static void main (String[] args)

int[] a = {10, 5, 20, 11, 6};

Arrays.sort(a); // sort by NSO

SOP (Arrays.binarySearch(a, 6)); => o/p: 1

SOP (Arrays.binarySearch(a, 14)); => o/p: -5

String[] s = {"A", "B", "C"};  
Arrays.sort(s);

SOP (Arrays.binarySearch(s, "B")); o/p: 2

SOP (Arrays.binarySearch(s, "S")); o/p: 3

Arrays.sort(s, new MyComparator());

SOP (Arrays.binarySearch(s, "A", new MyComparator())); // o/p: 0

SOP (Arrays.binarySearch(s, "S", new MyComparator())); // o/p: 1

SOP (Arrays.binarySearch(s, "N")); // unpredictable result

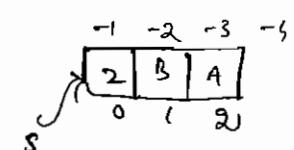
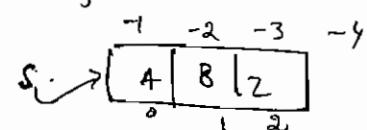
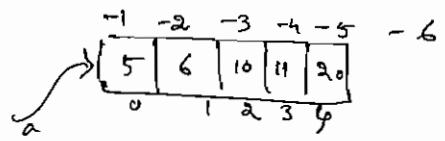
class MyComparator implements Comparator

{ public int compare (Object o1, Object o2)

String s1 = o1.toString();

String s2 = o2.toString();

return s2.compareTo(s1);



↳ Conclusion - Today's to List:

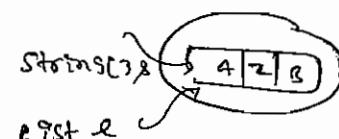
Arrays class containing asList() method for this

Public static List asList (Object[] a)

This method won't create an independent List object, just we are viewing existing array in List form

Ex: String[] s = {"A", "Z", "B"};

List l = Arrays.asList(s);



Conclusion:

① By using array reference we can perform any change automatically that change will be reflected to List reference.

Similarly using List reference if we perform any change automatically that change will be reflected to array.

② By using List reference we can't perform any operation which varies the size; otherwise we will get RuntimeException saying UnsupportedOperation exception

Ex:

l.add("K"); }  
l.remove(1); } RE: UnsupportedOperationException

l.set(1, "K");

③ By using List reference we can't replace heterogeneous objects, otherwise we will get RuntimeException saying ArrayStoreException

Ex:

import java.util.\*;  
class ArraysAsListDemo

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

String[] s = {"A", "Z", "B"};

List l = Arrays.asList(s);

Sop(l); o/p: [A, Z, B]

s[0] = "K";

Sop(l); o/p: [K, Z, B]

l.set(1, "L");

for(String s : s)

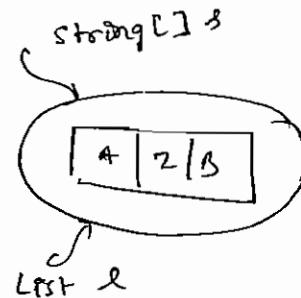
Sop(s); o/p: [K, L, B]

l.add("durga"); RE: UnsupportedOperationException

l.remove(2); RE: UnsupportedOperationException

l.set(1, new Integer(10)); RE: ArrayStoreException

3 Y



## Generics

- (1) Introduction
- (2) Generic classes
- (3) Bounded types
- (4) Generic methods and wild-card character (?)
- (5) Communication with Non-Generic code
- (6) Conclusions

### ① Introduction

The main objectives of Generics are to provide type-safety and to resolve type-casting problems.

#### Case 1:

##### Type-safety:

Arrays are type-safe i.e we can give the guarantee for the type of elements present inside Array.

Ex: if our programming requirement is to hold only String type of objects we can choose String[] array.

By mistake if we are trying to add any other type of objects we will get compilation error.

##### Ex:

```
String[] s = new String[10000];
```

```
s[0] = "durga";
```

```
s[1] = "Ravi";
```

```
s[2] = new Integer(10);
```

```
s[2] = "shiva";
```

C.e:

Incompatible types

found: java.lang.Integer

required: java.lang.String

Hence String[] array can contain only String type of objects due to this we can give the guarantee for the type of elements present inside Array 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 give the guarantee for the type of elements present inside collection

for example if our programming requirement is to hold only String type of Objects, by mistake if we choose ArrayList, by mistake if we are

trying to add any other type of object we won't get any compiletime error but the program may fail at runtime.

Eg:

```
ArrayList l = new ArrayList();
l.add("durga");
l.add("Ravi");
l.add(new Integer(10));
    |
    |
```

nameless  
Dictionary

1) Retrieving

```
String name1 = (String) l.get(0);
String name2 = (String) l.get(1);
String name3 = (String) l.get(2); X
Re: ClassCastException
```

Hence we can't give the guarantee for the type of elements present inside collection due to this collections are not safe to use with respect to type.  
i.e. Collections are not type safe.

Cases:

### Type-casting

In the case of arrays at the time of retrieval it is not required to perform type casting because there is a guarantee for the type of elements present inside array.

Eg:

```
String [] s = new String[1000];
s[0] = "durga";
    |
    |
```

{  
no  
guarantee in  
array elements  
(string-type.)}

```
String name1 = s[0];
```

type casting not required.

But in the case of collections at the time of retrieval compulsorily we should perform type-casting because here is no guarantee for the type of elements present inside collection.

Eg:

```
ArrayList l = new ArrayList();
l.add("durga");
```

String name = l.get(0);

String name = (String) l.get(0);

Type-casting is mandatory

Hence typecasting is a bigger headache in collections.

→ To overcome above problems of Collections Sun People introduced Generics Concept in 1.5 version

Hence the main objectives of Generics are

- (1) To provide Type-safety
- (2) To resolve Type-casting problems

for example to hold only String type of objects we can create Generic Version of ArrayList object as follows.

`ArrayList<String> l = new ArrayList<String>();`

Sec-2

for this ArrayList we can add only String type of objects By mistake if we are trying to add any other type then we will get compiletime error.

Eg: `l.add("durga");` ✓

`l.add("Ravi");` ✓

`l.add(new Integer(10));` → C.E

`l.add("shiva");` ✓

Hence through Generics we are getting type-safety

→ At the time of Retrieval we are not required to perform type-casting

Eg:

`ArrayList<String> l = new ArrayList<String>();`

`l.add("durga");`

⋮

`String name = l.get(0);`

Type casting is not required

Hence through Generics we can solve type-casting Problem.

Causes

Incompatible types

found: j.l.Object

required: j.lang.String

(14A)

`ArrayList l = new ArrayList();`

- ① It is a non Generic version of ArrayList object.
- ② For this ArrayList we can add any type of Object and hence it is not type safe.
- ③ At the time of retrieval compulsory we have to perform type casting.

`ArrayList<String> l = new ArrayList<String>();`

- ① It is a Generic version of ArrayList object.
- ② For this ArrayList we can add only String type of Objects and hence it is typesafe.
- ③ At the time of Retrieval we are not required to perform type-casting.

### Conclusion ①

Polyorphism concept Applicable only for the Base type but not for Parameter type. (Usage of Parent reference to hold child object) is the concept of Polymorphism.

Eg:

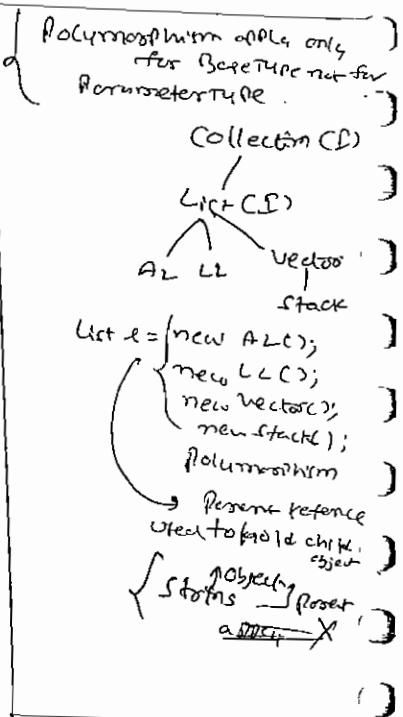
```
ArrayList<String> l = new ArrayList<String>();
```

Base type                          Parameter type

```
List<String> l = new ArrayList<String>();
```

```
Collection<String> l = new ArrayList<String>();
```

```
X ArrayList<Object> l = new ArrayList<String>();
```



C.E: Incompatible types  
 found: ~~ArrayList<String>~~ <java.lang.String>  
 required: ArrayList<Object> <java.lang.Object>

### Conclusion ②

For the type parameter we can provide any class or interface name but not primitives.

If we are trying to provide primitive then we will get Compiletime error.

Eg: `ArrayList<int> m = new ArrayList<int>();`

C.E: unexpected types  
 found & int  
 required & reference

## Q) Generic classes

Until 1.4 version a non Generic version of ArrayList class is declared as follows

```
Class ArrayList
{
    add(Object o)
    Object get(int index)
}
```

The argument to add() method is object and hence we can add any type of object to the ArrayList due to this we are missing type safety.

The return type of get() method Object hence at the time of retrieval we have to perform type-casting.

\*→ But in 1.5 version a Generic version of ArrayList class is declared as follows

```
Class ArrayList<T> Type Parameter
{
    add(T t)
    T get(int index)
}
```

Based on our runtime requirement T will be replaced with our provided type.  
for example to hold only String type of objects a Generic version of ArrayList object can be created as follows

`ArrayList<String> e = new ArrayList<String>();`  
for this requirement compiler consider version no 1.5 of ArrayList class as follows.

```
Class ArrayList<String >
{
    add(String s)
    String get(int index)
}
```

The argument to add() method is String type hence we can add only String type or Objects by mistake if we are trying to add any other type we will get Compiletime error.

Eg:

`l.add("durga");`

`l.add(new Integer(10));`

↳ Can not find symbol

Symbols: method add (g. l. Integer)  
location: class ArrayList <String>

Hence through Generics we are getting type-safety

The return type of retrieval we are not required to perform type-casting

Eg:

String name = l.get(0);

type casting is not required

In Generics we are associating a type parameter to the class such type of Parameterized classes are nothing but Generic classes (or) template classes

{  
C++  
Template  
Type  
Generics  
}

→ Based on our requirement we can define our own Generic classes also

Ex:

Class Account <T>  
{  
    }  
    {  
        }

Account <Gold> a1 = new Account <Gold>();

Account <Platinum> a2 = new Account <Platinum>();

Ex: //Create our own Generic class

Class GenLT >  
{  
    T obj;  
    Gen LT obj  
    {  
        this.obj = obj;  
    }  
}

```

public void show()
{
    System.out.println("The type of ob;" + ob.getClass().getName());
}

public T getOb()
{
    return ob;
}

class Grendemo
{
    public static void main(String[] args)
    {
        Grendemo g1 = new Grendemo("durga");
        g1.show();           // The type of ob: java.lang.String
        System.out.println(g1.getOb());  durga

        Grendemo g2 = new Grendemo(10);
        g2.show();           // The type of ob: java.lang.Integer
        System.out.println(g2.getOb());  10

        Grendemo g3 = new Grendemo(10.5);
        g3.show();           // The type of ob: java.lang.Double
        System.out.println(g3.getOb());  10.5
    }
}

```

{  
  Taming type  
}

### (3) Bounded Types

We can bound the range by using Type Parameter extends keyword for a particular such types are called Bounded types.

Ex: class Test<T>

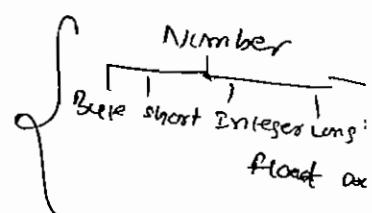
}      =  
Unbounded type.

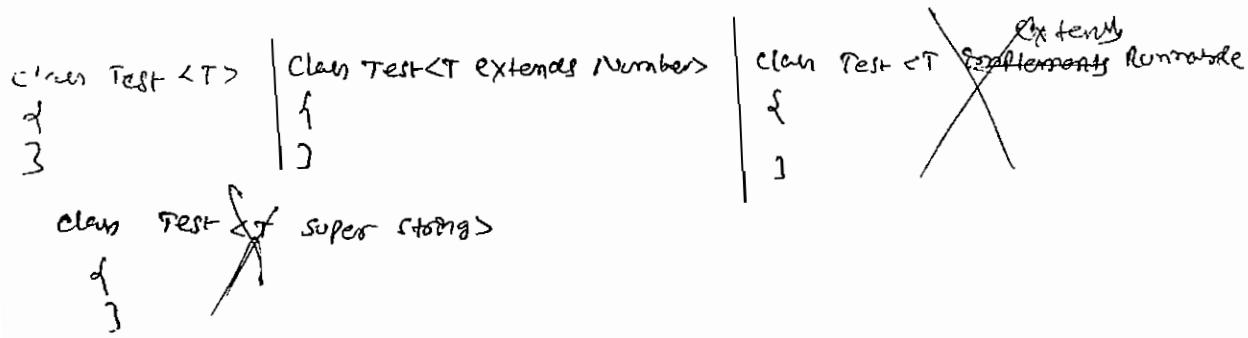
As the type parameter we can pass any type and there are no restrictions and hence it is

```

Test<Integer> t1 = new Test<Integer>();
Test<String> t2 = new Test<String>();

```





### Syntax for Bounded Type

```
class Test < T extends X >
{
    ...
}
```

X can be either class or interface. If X is a class then as the type parameter we can pass either X type or its child classes.  
 If X is an interface then as the type parameter we can pass either X type or its implementation classes.

Ex ① `class Test < T extends Number >`

```
d
  ...
y
```

```
Test < Integer > t1 = new Test < Integer >();
```

```
Test < String > t2 = new Test < String >();
```

C.e

Type parameter java.lang.String is not in its bound

param (and)

②

`class Test < T extends Runnable >`

```

  ...
}
```

```
Test < Runnable > t1 = new Test < Runnable >();
```

```
Test < Thread > t2 = new Test < Thread >();
```

```
Test < Integer > t3 = new Test < Integer >();
```

C.e

Type parameter java.lang.Integer is not with in its bound

We can define bounded types even in combination also

Ex 8 ①

Class Test<T> extends Number & Runnable  
{  
}

as the type parameter we can take anything which should be child class of Number and should implement Runnable interface.

✓ ② Class Test<T> extends Runnable & Comparable

✗ ③ Class Test<T> extends Number & Runnable & Comparable

Class Test<T> extends Runnable & Number

(Because we have to take class first followed by interface next)

✗ ④ Class Test<T> extends Number & Thread

(Because we can't extend more than one class simultaneously)

### Conclusion

Note:

① We can define bounded types only by using extends keyword and we can't use implements and super keywords but we can replace implements keyword purpose with extends keyword.

Class Test<T> extends Number ✓ | Class Test<T> implements Runnable  
{ } | {  
} | X  
}

Class Test<T> extends Runnable  
{ } ✓ | Class Test<T> super String  
{ } =  
} X

② Has the type parameter 'T' but it's convention to use 'T', we can take any valid Java Identifier

Class Test<T> | Class Test<x> | Class Test<A> | Class Test<durga>  
{ } | { } | { } | { }  
Type Parameter | } | } | }  
✓ | ✓ | ✓ | ✓ |  
} | } | } | }  
} | } | } | }

③ Based on our requirement we can declare any number of type parameters and all these type parameters should be separated with comma (,)

```

class Test<A,B>    class Test<X,Y,Z>
{
}
}

class HashMap<K,V>  ↗ keyType
{
}
}

HashMap<Integer, String> h = HashMap<Integer, String>(); ↗ valueType

```

④ Generic Methods and wild-card characters (?) See ④

①  $m_1(\text{ArrayList<String>} l)$ :

(i) we can call this method by passing ArrayList of only String type.

(ii) But with in the method we can add only String type or Objects to the list by mistake if we are trying to add any other type item we will get compiletime error

Ex  $m_1(\text{ArrayList<String>} l)$

```

{
    l.add("A");
    l.add(null);
    l.add(10); X
}

```

```

AL<String> l = new AL<String>();
l.add("A");
l.add(null);
l.add(10); X

```

how to call  
what activity needs

②  $m_1(\text{ArrayList<} ? \text{>} l)$

(i) we can call this method by passing ArrayList of any unknown type

(ii) But within the method we can't add anything to the list except null because we don't know the type exactly.

(67/1)

null is allowed because it is valid value for any type.

Sx8      m1(ArrayList<?> l)

l.add(10.5); X  
l.add('A'); X  
l.add(10); X  
l.add(null); ✓

(Suppose we pass ArrayList<String> l  
+ 10.5 X  
invalid)

→ This type of methods are best suitable for read only operation

### (3) m1(ArrayList<?> extends X > l)

{  
  m1<ArrayList>l  
  l.add(l);  
}

(i) X can be either class or interface

(ii) If X is a class then we can call this method by passing ArrayList of either X type or its child classes

If X is an interface then we can call this method by passing ArrayList or either X type or its implementation classes

But with this method we can't add anything to the list except null because we don't know the type (of X) exactly.

This type of methods also best suitable for read only operation.

### (4) m1(ArrayList<? super X > l)

(i) X can be either class or interface

(ii) If X is a class then we can call this method by passing ArrayList of either X type or its super classes.

(iii) If X is an interface then we can call this method by passing ArrayList of either X type or super class of implementation class of X.

\* → But with this method we can add X type of objects and null to the list

Object Runner  
of Thread

l.add(X);  
  X type  
l.add(null); ✓

Q1

- ① `ArrayList<String> l = new ArrayList<String>();` ✓
- ② `ArrayList<?> l = new ArrayList<String>();` ✓
- ③ `ArrayList<?> l = new ArrayList<Integer>();` ✓
- ④ `ArrayList<? extends Number> l = new ArrayList<Integer>();` ✓
- ⑤ `ArrayList<? extends Number> l = new ArrayList<String>();` ✗

C. e.g.  
incompatible types  
found: `ArrayList<String>`  
required: `ArrayList<? extends Number>`

- ⑥ `ArrayList<? super String> l = new ArrayList<Object>();`
- ⑦ `ArrayList<?> l = new ArrayList<Object>();`
- ⑧ `ArrayList<?> l = new ArrayList<? extends Number>();`  
*(we can't use ? mark here)*

C. e.g.  
unexpected type  
found: ?  
required: class or interface  
without bound

C. e.g.  
unexpected type  
found: ? extends Number  
required: class or interface without bound

→  
no

```
import java.util.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        ArrayList<?> l = new ArrayList<String>();  
    }  
}
```

Valid

(2) We can declare type parameter either at Class Level or at Method Level

Declaring type parameter at class Level:

Class Test<T>

{

We can use 'T'  
with this class  
based on our requirement

}

Declaring type parameter at method Level:

Class Test

We have to declare type parameter just before return type

Ex:

Class Test

{

Public <T> void m1(T ob)

{

We can use 'T' anywhere

within this method based on our requirement

} }

\* We can define bounded types even at method level also

Ex:

Public <T> void m1()

↓

✓ <T extends Number>

✓ <T extends Runnable>

✓ <T extends Number & Runnable>

✓ <T extends Comparable & Runnable>

✓ <T extends Number & Comparable & Runnable>

✗ <T extends Runnable & Number> { first we have to take class and then interface }

✗ <T extends Number & Thread> → { we can't extend more than one class }

## (5) Communication with non-Generic code

If we send Generic object to non Generic area then it starts behaving like non-Generic object.

Similarly if we send non-Generic object to Generic area then it starts behaving like Generic object.

i.e. The location in which object present based on that behaviour will be defined

Exe Class Test

```

P = v main (String[] args)
{
    ArrayList<String> l = new ArrayList<String>();
    l.add ("durga");
    l.add ("Ravi");
    l.add (10); → CE
    m (l);
    System.out.println (l);
    l.add (10.5); → CE
}
  
```

Generic Area.

P = v m (ArrayList l)

```

l.add (10);
l.add (10.5);
l.add (true); } non-generic area
  
```

## Conclusion

① The main purpose of Generics is to provide type safety and to resolve type-casting problems. Type safety and type-casting both are applicable at compiletime hence Generics concept also applicable only at compiletime but not at runtime.

→ at the time of compilation as last step Generic syntax will be removed. and hence for the JVM Generic syntax won't be available.

At the time of compilation Generic concept removed  
→ JVM always goes to call runtime object  
Compiler check → referencing type.

→ Hence the following declarations are equal.

```
ArrayList l = new ArrayList<String>();  
ArrayList l = new ArrayList<Integer>();  
ArrayList l = new ArrayList<Double>();  
ArrayList l = new ArrayList();
```

AL l = new AL<String>  
↓ <Integer  
here references are equal.

Ex:

```
ArrayList l = new ArrayList<String>();  
l.add(10);  
l.add(10.5);  
l.add(true);  
System.out.println(l); [10, 10.5, true].
```

The following declarations are equal.

```
ArrayList<String> l = new ArrayList<String>();  
ArrayList<String> l = new ArrayList();
```

Left hand reference equal

for these ArrayList Objects we can add only String type of objects

Ex: Class Test

```
public void m1(ArrayList<String> l) {  
}
```

```
public void m1(ArrayList<Integer> l) {  
}
```

C. E. G.

name clash: Both methods have same signature

At compile time

- ① compile code normally by considering generic syntax
- ② Remove Generic syntax
- ③ compile once again resultant code.

Class Test  
{  
 public void m1(int i) {  
 m1(i);  
 }  
 public int m1(int i) {  
 return i;  
 }  
}  
With m1 as a class  
2 methods with a  
same signature  
m1(int), is not allowed  
Test defined in oops

1  
2  
3

7  
8  
9

0  
1  
2

3  
4  
5

6  
7  
8

9  
0  
1

2  
3  
4

5  
6  
7

8  
9  
0

1  
2  
3

## Inner Classes

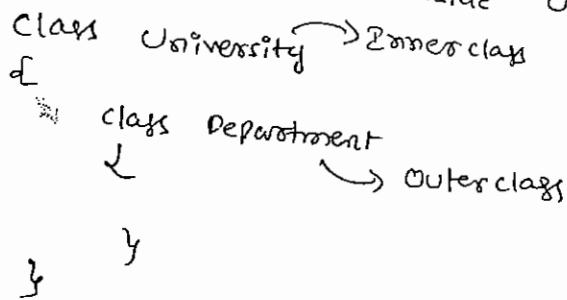
(168)

Sometimes we can declare a class inside another class such type of classes are called inner classes.

- Inner classes concept introduced in 1.1 version to fix GUI bugs as the part of Event handling but because of powerful features and benefits of inner classes slowly programmers started using in regular coding also.
- without existing one type of object if there is no chance of existing another type of object then we should go for inner classes.

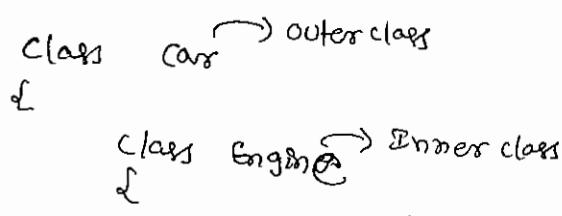
Ex1:

University consists of several departments without existing university there is no chance of existing department hence we have to declare Department class inside University class.



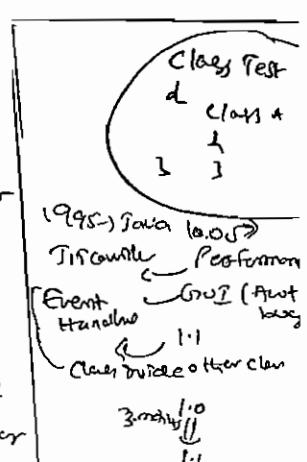
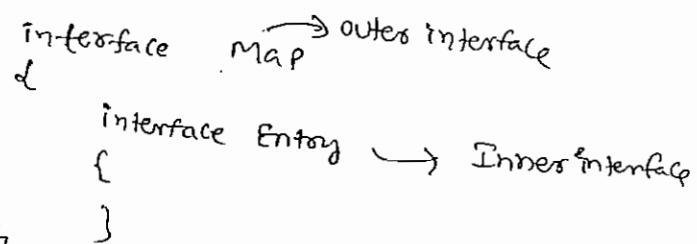
Ex2:

without existing Car Object there is no chance of existing Engine Object hence we have to declare Engine class inside Car class.



Ex3:

Map is a group of key value Pairs and each key value pair is called an Entry without existing Map object there is no chance of existing Entry object hence interface Entry is defined inside map interface.



key	value
101	durga
102	Ravi
103	Shiva
104	Pawan

map

Ques

① Without existing outer class object there is no chance of existing inner class object

② The relation b/w outer class and inner class is not Is-A relation and it is Has-A relationship (composition or aggregation) (University Has-A Department)

→ Based on position of declaration and behaviour all inner classes are divided into 4 types

① Normal (or) Regular Inner classes

② Method Local Inner classes

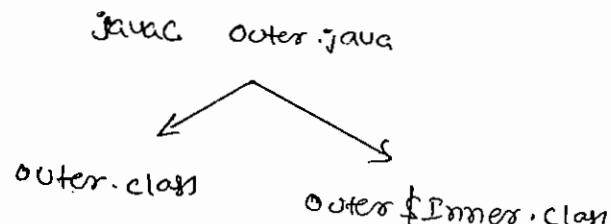
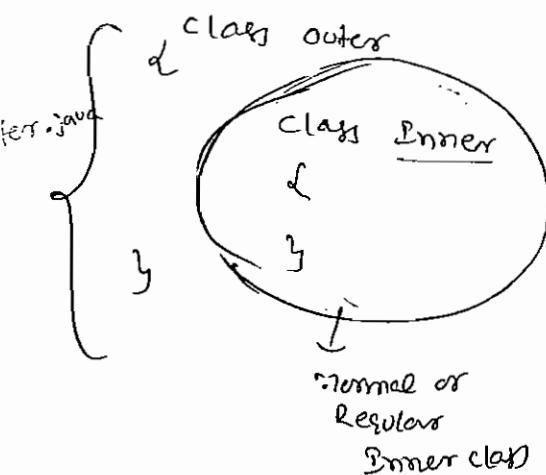
③ Anonymous Inner classes

④ Static nested classes

① Normal (or) Regular Inner classes

If we are declaring any named class directly inside a class without static modifier such type of inner class is called Normal (or) Regular inner class.

Ex:

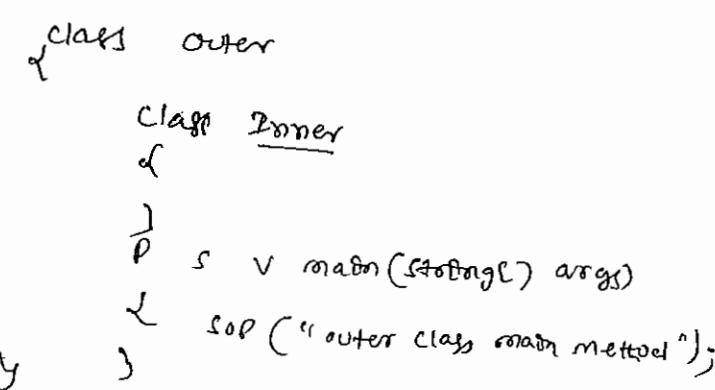


java Outer

RE: NoSuchMethodError: main

java Outer\$Inner

RE: NoSuchMethodError: main



java Outer.java

OuterClass Outer\$InnerClass

java Outer

RE: Outer class main method

java Outer\$Inner

RE: inner class main method

Ex 3: Class Outer

{

Class Inner

{

P = v main (String[] args)

{

System.out.println ("Inner class main method");

Inner classes can not have static declarations

C-E:

→ Inside inner class we can't declare any static members hence we can't declare main method and we can't run inner class directly from command prompt.

Ex:

~~javac Outer.java~~  
~~Outer-class Outer.java~~  
~~Outer-class Outer\$Inn~~  
~~cl.~~

With out existing outer class there is no chance of existing inner class don't talk to inner classes directly →



You can't come here dire

Case 1

Accessing inner class code from static area of outer class.

Class Outer

{ Class Inner

    { public void m1()

        System.out.println ("Inner class method");

    }

    P = v main (String[] args)

    { Outer o = new Outer();

    Outer.Inner i = o.new Inner();

    i.m1();

New Outer

new Outer().new Inner().m1();

To call m1()  
 ↓  
 inner class  
 ↓ require  
 with out exist  
 outer class  
 no inner class  
 object

Outer

Outer.Inner i =

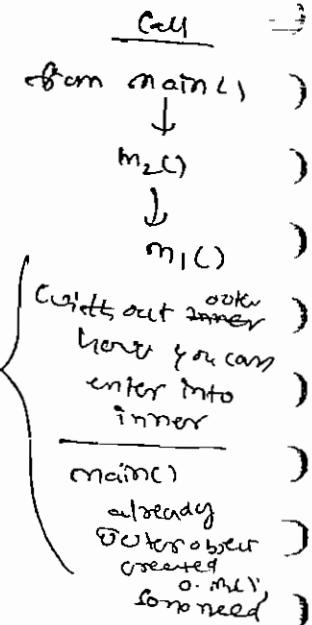
new Outer().new Inner()

Case 2: Accessing Inner class code from instance Area of outer class

```
class Outer
{
    class Inner
    {
        public void m1()
        {
            System.out.println("Inner class method");
        }

        public void m2()
        {
            Inner i = new Inner();
            i.m1();
        }

        public static void main(String[] args)
        {
            Outer o = new Outer();
            o.m2();
        }
    }
}
```



Case 3:

Accessing Inner class code from outside of Outer class

```
class Outer
{
    class Inner
    {
        public void m1()
        {
            System.out.println("Inner class method");
        }

        public static void main(String[] args)
        {
            Outer o = new Outer();
            Outer.Inner i = o.new Inner();
            i.m1();
        }
    }
}
```

for

```
+ to call m1()  
  Outer  
  create  
  but without  
  Outer can't create  
  Inner so  
  create outer object  
  Inner object } )
```

## Accessing Inner class code

from Static Area of Outer class  
(or)

from outside of outer class

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.m1();
```

from Instance Area of Outer class

```
Inner i = new Inner();
i.m1();
```

\* From Normal (or) Regular inner class we can access both static and non static members of outer class directly.

Ex: Class Outer

```
int x=10;
static int y=20;
```

o/p: 

10
20

```
class Inner
{
    public void m1()
    {
        System.out.println(x);
        System.out.println(y);
    }
}
```

```
P = > main(String[] args)
{
```

new Outer().new Inner().m1();

Sec-2  
Normal regular inner classes  
we can't declare static members  
but we can access

\* With in the inner class this always refers current inner class object if we want to refer current outer class object we have to use Outerclassname.this

Ex: class Outer

```
{ int x=10;
    class Inner
    {
        int x=100;
        public void m1()
        {
            int x=1000;
            System.out.println(x);
            System.out.println(this.x); // System.out.println(Inner.this.x); 100
        }
    }
```

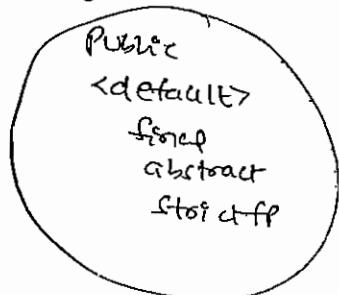
```
2 3 System.out.println(Outer.this.x); 100 ✓
```

```

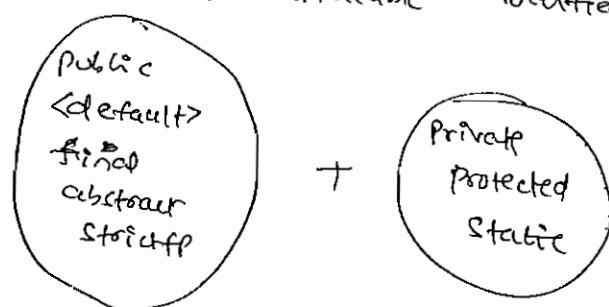
P. s v main(String[] args)
{
    new Outer().new Inner().m1();
}

```

\* The only applicable modifiers for outer classes are



But for inner classes applicable modifiers are



### Nesting of Inner Classes

In inner inner class we can declare another inner class i.e nesting of inner classes is possible

class A  
d  
 class B  
 d  
 class C  
 d  
 public void m1()  
 {  
 System.out.println("Innermost class method");  
 }  
 }  
 }  
}

{  
 Compile error  
 A, B (B, C)  
 C1 B we can't  
 call m1();  
}

class Test  
d  
 P  
 s v main(String[] args)

A a = new A();

A.B b = a.new B();

A.B.C c = b.new C();  
c.m1();

## ② Method Local Inner classes

(169)

Some times we can declare a ~~method~~ class inside a method such type of inner classes are called method local inner classes.

- The main purpose of method local inner class is to define method specific repeatedly required functionality.
- Method local inner classes are best suitable to meet nested method requirement.
- We can access method local inner classes only with in the method where we declared.
- Outside of the method we can't access. because of its less scope. Method local inner classes are most rarely used type of inner classes.

Same function  
repeatedly used  
in our method  
declare method at  
class level.  
m1  
{ int x=10;  
}

Exs

```
Class Test
{
    Public void m1()
    {
        Class Inner
        {
            Public void sum(int x, int y)
            {
                System.out.println("The sum is " + (x+y));
            }
        }
    }
}
```

Inner i = new Inner();

i.sum(10, 20);  
};

i.sum(100, 200);  
};

i.sum(1000, 2000);  
};

Output:  
The sum: 30  
The sum: 300  
The sum: 3000

↓ ↓      v main(String[] args)

Test t = new Test();

t.m1(); → instance method

→ We can declare static methods inside both instance and

- If we declare inner class ~~directly inside a method~~ inside instance method then from that method local inner class we can access both static and non static members of outer class directly.
- If we declare inner class inside static method then we can access only static members of outer class directly from that method local inner class.

Exe

```

class Test
{
    int x=10;
    static int y=20;
    public void m1()
    {
        class Inner
        {
            public void m1()
            {
                System.out.println(x); → ①
                System.out.println(y); → ②
            }
        }
        Inner i = new Inner();
        i.m1();
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

If we declare  $m_1()$  method as static (Public static void  $m_1()$ ) then at line ① we will get compile time error saying non static variable  $x$  can not be reference from a static context.

\* From method local inner class method in which we declare inner class we can't access local variables of the if the local variable declared as final then we can access.

Exe

```

class Test
{
    public void m1()
    {
        final int x=10;
        class Inner
        {
            public void m2()
            {
                System.out.println(x);
            }
        }
        Inner i = new Inner();
        i.m2();
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

(Q8) local variable  $x$  is accessed from within inner class; needs to be declared final. (17)

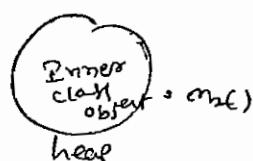
If we declare  $x$  as final (final int  $x=10$ ) then we won't get any compiletime error

a will be created inside stack memory



object will create at heap area

it is an object or Inner class



i.  $m1()$  → inner class object  
we call  $m1()$  method

once  $m1()$  method local variable completely destroyed  
or after completing  $m1()$  there may be chance of object present in heap

if i call  $m2()$  method again on that object on inner class object.  $m2()$  or is destroyed already from where i can get so

Q8: we get .

→ declare Local Variable as final why there is no final?

If we declare final every final variable replaced value at compiletime so  
 $x$  value is ~~not~~ 10       $m2()$       } if we call  $m2()$  then we get o/p as 10  
                  {  $sop(10);$  }

(Q1) Consider the following code?

Class Test

d int p=10;

static int q=20;

public void m1()

d int k=30;

final int m=40;

Class Inner

{ Public void m2()

{ Line ① }

} y }

{ i, j m1() method  
so when both instance & static members  
K X Local variables  
m / c

At line ① which of the following variables we can access directly

- i ✓
- j ✓
- K X
- m ✓

Q2) If we declare `m1()` method of static then at Line ① which variables we can access directly?

Class Test

```
int i=10;  
static int j=20;  
public static void m1()  
{  
    int k=30;  
    final int m=40;  
    class Inner  
    {  
        public void m2()  
        {  
            Line ①  
        }  
    }  
}
```

① X  
j ✓  
② X  
m ✓

Q3) If we declare `m2()` method of static then at Line ① which variables we can access directly?

Class Test

```
int i=10;  
static int j=20;  
public void m1()  
{  
    int k=30;  
    final int m=40;  
    class Inner  
    {  
        public static void m2()  
        {  
            Line ①  
        }  
    }  
}
```

j  
k  
m

We will get compilation error because we can't declare static members inside inner classes.

The only applicable modifiers for method Local  
inner classes are

final  
abstract  
strictfp

If we are trying to apply any other modifier then we will get compilation error.

$m_1()$   
final ( )  
Only applicable modifier  
for Local variable is  
final.

### ③ Anonymous Inner Classes

Sometimes we can declare inner class without name such type of inner classes are called Anonymous inner classes.

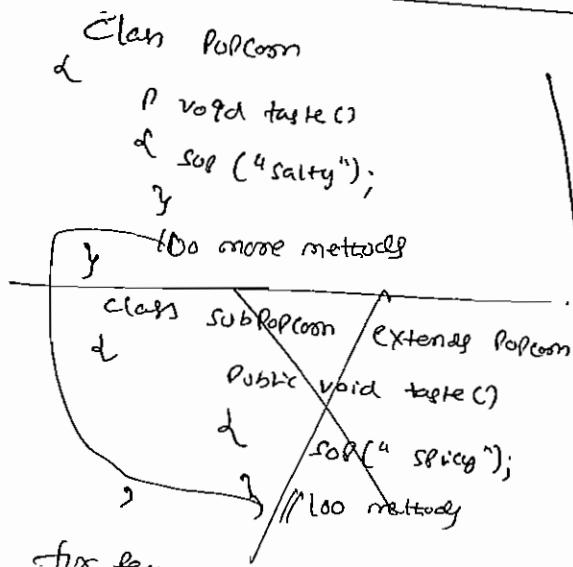
→ The main purpose of Anonymous inner classes is just for ~~instance~~ use (one time usage)

Base on declaration and behaviour there are 3 types of anonymous inner classes

- ① Anonymous Inner class that extends a class
- ② Anonymous Inner class that implements an interface
- ③ Anonymous Inner class that defined inside arguments

add or  
Concrete  
build  
a per.  
Contractor  
Conduct  
none  
mobile  
future we can't  
create him

#### ① Anonymous Inner Class that extends a class



```

class Test {
    void main(String[] args) {
        Popcorn p = new Popcorn();
        p.taste(); // salty
    }
}

```

```

Popcorn p=newPopcorn();
Popcorn p=new Popcorn();
{
    Thread t=new Thread();
    t.start();
    Parent reference
    Runnable r=new Runnable();
    r.run();
    Executing a class
    that extends
    Popcorn without na.
    & create child classes
}

```

class files generated → Popcorn.class  
Test.class  
for anonymous class → 1.class file generated  
Test 1. class

Top Level Class

1st anonymous class in test  
Test 1. class

( Sop ( p.genClass().getName() )  
to print class names

overridable  
method  
Come into  
exist

① Anonymous Inner class that extends a class:

Class Popcorn

{

    public void taste()

{

        System.out.println("salty");

}

}

Class Test

{

    public static void main(String[] args)

{

        Popcorn p = new Popcorn()

{

            public void taste()

{

                System.out.println("spicy");

}

}

        p.taste(); // spicy

    Popcorn p1 = new Popcorn();

    p1.taste(); // salty

    Popcorn p2 = new Popcorn()

{

            public void taste()

{

                System.out.println("sweet");

}

}

        p2.taste(); // sweet

    // I want sweet

    // I don't want salty &

    // spicy

    System.out.println(p.getClass().getName()); // Test \$ 1

    System.out.println(p1.getClass().getName()); // Popcorn

    System.out.println(p2.getClass().getName()); // Test \$ 2

    ↳ Outer class name

The generated .class files are

Popcorn.class

Test.class

Test \$ 1.class

Test \$ 2.class

↳ 1st anonymous class

↳ 2nd anonymous class

## Analysis

(13)

(1) `Popcorn P = new Popcorn();`

Just we are creating Popcorn object

(2) `Popcorn P = new Popcorn();`

{

i) we are declaring a class that extends Popcorn without name (anonymous inner class)

ii) for that child class we are creating an object with parent reference

(3) `Popcorn P = new Popcorn();`

{

public void taste()

{ sop ("spicy");

}

i) we are declaring a class that extends Popcorn without name (Anonymous inner class)

ii) in that child class we are overriding taste() method.

iii) for that child class we are creating an object with parent reference.

## Defining a Thread by extending Thread class

### Normal class Approach

refining a thread

```

class MyThread extends Thread
{
    public void run()
    {
        for (int i=0; i<10; i++)
        {
            sop ("child thread");
        }
    }
}

class ThreadDemo
{
    public static void main (String [] args)
    {
        MyThread t = new MyThread();
        t.start();
        for (int i=0; i<10; i++)
        {
            sop ("main thread");
        }
    }
}

```

### Anonymous Innerclass Approach

```

class ThreadDemo
{
    public static void main (String [] args)
    {
        Thread t = new Thread()
        {
            public void run()
            {
                for (int i=0; i<10; i++)
                {
                    sop ("child thread");
                }
                t.start();
                for (int i=0; i<10; i++)
                {
                    sop ("main thread");
                }
            }
        };
    }
}

```

If the job is temporary just one time usage no need ob going top level class

② Anonymous inner class that implements an interface.

Defining a Thread by implementing Runnable Interface.

Normal class approach

```
class myRunnable implements Runnable  
{  
    public void run()  
    {  
        for(int i=0; i<10; i++)  
        {  
            System.out.println("child thread");  
        }  
    }  
}  
  
class ThreadDemo  
{  
    public static void main(String[] args)  
    {  
        myRunnable r = new myRunnable();  
        Thread t = new Thread(r);  
        t.start();  
        for(int i=0; i<10; i++)  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

Anonymous inner class approach

```
class ThreadDemo  
{  
    public static void main(String[] args)  
    {  
        Runnable r = new Runnable()  
        {  
            public void run()  
            {  
                for(int i=0; i<10; i++)  
                {  
                    System.out.println("child thread");  
                }  
            }  
        };  
        Thread t = new Thread(r);  
        t.start();  
        for(int i=0; i<10; i++)  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

③ Anonymous inner class that define inside arguments

class ThreadDemo

```
{  
    public static void main(String[] args)  
    {  
        Thread t = new Thread(new Runnable()  
        {  
            public void run()  
            {  
                for(int i=0; i<10; i++)  
                {  
                    System.out.println("child thread");  
                }  
            }  
        });  
        t.start();  
        for(int i=0; i<10; i++)  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

(if t.start(), )

## Normal Java class Vs Anonymous Inner class

- ① A normal java class can extend only one class at a time whereas anonymous inner class also can extend only one class at a time
- ② A normal java class can implement any number of interfaces simultaneously but anonymous inner class can implement only one interface at a time
- ③ A normal java class can extend ~~one~~ a class and can implement any number of interfaces simultaneously but anonymous inner class can extend a class (or) can implement an interface but not both.
- ④ In normal java class we can write any number of constructors simultaneously. but in anonymous (because the name of the class and name of the constructor must be same) but anonymous inner classes not having any name

Note:

- if the requirement is standard and required several times then we should go for normal top level class.
- If the requirement is temporary and required only once (Instant use) then we should go for anonymous inner class.
- we can use anonymous inner classes frequently
- In GUI based applications to implement Event Handling

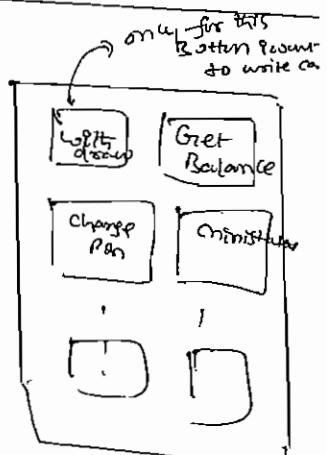
Eg:

```
Class MyGUiFrame
{
    JButton b1, b2, b3, b4, b5, b6;
    ..
}
```

b. add ActionListener(new ActionListener()

```
{
    public void actionPerformed
        (ActionEvent e)
    {
        // b1 specific functionality
    }
}
```

```
Thread t = new Thread()
{
    public void run()
    {
        public class extends Runnable
        {
            public void run()
            {
                name();
            }
        }
    }
}
```



Group form  
ActionListener  
↳ interface  
# no class myAction Listener interface  
X no need

b2. addActionListener (new ActionListener())

Inner class implements ActionListener

```

    {
        public void actionPerformed(ActionEvent e)
        {
            // by specific functionality
        }
    }
}

```

not required  
to take  
separate follow  
class.

Ex:

```

import java.awt.*;
import java.awt.event.*;
public class TestDemo
{
    public static void main(String[] args)
    {
        Frame f = new Frame();
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                for (int i=1; i<=10; i++)
                    System.out.println("I am closing window : " + i);
                System.exit(0);
            }
        });
        f.add(new Label("I can create Executable jar file!"));
        f.setSize(500, 500);
        f.setVisible(true);
    }
}

```

## ④ Static nested classes

→ Some times we can declare inner class with static modifier  
Such type of inner classes are called static nested classes

In the case of Normal or regular  
inner class with out existing outer class  
Object there is no chance of existing  
inner class object i.e. inner class  
object is strongly associated with  
outer class object

class Test
{
 static int x=10; → without existing
 Test object
 There is no
 chance of
 existing
 (x)
}

Static int y=20;

| class hence  
no related to  
object

Nested  
↓ no strong association

But in the case of static nested classes without existing outer class object there may be a chance of existing nested class object hence static nested class object is not strongly associated with outer class object.

Ex:

```

class Outer {
    static class Nested {
        public void m1() {
            System.out.println(" static nested class method");
        }
    }
    public static void main(String[] args) {
        Nested n = new Nested();
        n.m1();
    }
}

```

*With in it same class we can access static mem directly*

If we want to create Nested class object from outside of outer class then we can create as follows

Ref `Outer.Nested n = new Outer.Nested();`

*outer class*

*class name.*

② In Normal (or) regular inner classes we can't declare any static members. But in static nested classes we can declare static members including static method hence we can invoke static nested class directly from command prompt.

Ex:

```

class Test {
    static class Nested {
        public static void main(String[] args) {
            System.out.println(" static nested class main method");
        }
        public static void main(String[] args) {
            System.out.println(" outer class main method");
        }
    }
}

```

javac Test.java  
 java Test  
 Java Test  
 -> Opt: outer class main method  
 -> static nested class main method.

from normal or regular inner classes we can access both static and non static members of outer class directly but from static nested classes we can access static members of outer class directly and we can't access non static members.

Exe

```
class Test
{
    int x=10;
    static int y=20;

    static class Nested
    {
        public void m1()
        {
            System.out.println(x); // C.E
            System.out.println(y);
        }
    }
}
```

non static variable x cannot be referenced from a static context

Differences b/w Normal (or) Regular inner class and Static nested class

Normal (or) Regular inner class	Static nested class
① without existing outer class object there is no chance of existing inner class object i.e. inner class object is strongly associated with outer class object.	① without existing outer class object there may be a chance of existing static nested class object <del>because</del> i.e. static nested class object is not strongly associated with outer class object.
② In normal (or) regular inner class we can't declare static members	② In static nested classes we can declare static members
③ In normal or regular inner class we can't declare main method and hence we can't invoke inner class directly from command prompt	③ In static nested classes we can declare main method and hence we can invoke nested class directly from command prompt
④ From normal or regular inner classes we can access both static and non static members of outer class directly	④ From static nested classes we can access only static members of outer class.

Various combinations of Nested classes and Interfaces.

### Case 1: Class inside a class

Without existing one type of object if there is no chance of existing another type of object then we can declare a class inside a class.

Eg: university consists of several departments without existing university there is no chance of existing Department, hence we have to declare Department class inside University class.

```
Class University
{
    Class Department
    {
    }
}
```

### Case 2:

### Interface inside a class

Inside a class if we require multiple implementations of an interface and all these implementations are related to a particular class then we can define interface inside a class.

Eg:

```
Class VehicleTypes
{
    Interface Vehicle
    {
        Public int getNoOfWheels();
    }

    Class Bus implements Vehicle
    {
        Public int getNoOfWheels()
        {
            Return 6;
        }
    }

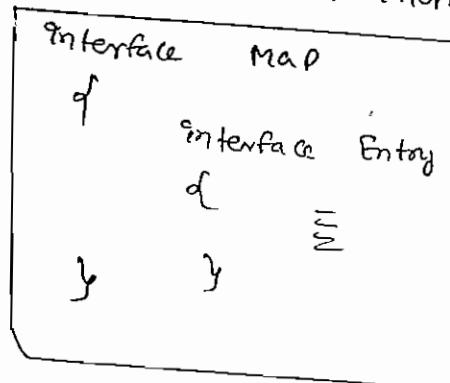
    Class Auto implements Vehicle
    {
        Public int getNoOfWheels()
        {
            Return 3;
        }
    }
}
```

### Case 36 Interface Inside Interface

We can declare interface inside interface

Eg: A Map is a group of key value pairs and each key value pair is called an Entry without existing Map object there is no chance of existing Entry object hence interface Entry is defined inside Map (I) interface.

key	value
lo1	durga
lo2	Ravi
lo3	Shiva
lo4	Pawan



→ Every interface present inside interface is always public and static whether we are declaring or not. hence we can implement inner interface directly without implementing outer interface. similarly whenever we are implementing outer interface we are not required to implement inner interface i.e. we can implement outer and inner interfaces directly independently.

Ex:

```

interface Outer {
    public void m1();
}

interface Inner {
    public void m2();
}

class Test implements Outer {
    public void m1() {
        System.out.println("Outer interface method implementation");
    }
}

class Testa implements Outer, Inner {
    public void m2() {
        System.out.println("inner interface method implementation");
    }
}
  
```

```

class Test
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.m1();
        Test t2 = new Test();
        t2.m2();
    }
}

```

### Case 4G Class inside Interface

If functionality of a class is closely associated with interface then it is highly recommended to declare class inside interface.

Ex: ① Interface EmailService

```

{
    public void sendMail(EmailDetails e);
}

class EmailDetails
{
    String to_list;
    String cc_list;
    String subject;
    String body;
}

```

In the above example and we are not using anywhere else hence EmailDetails class is recommended to declare inside EmailService interface.

We can also ~~implement~~ provide default implementation for that interface.

Ex: Interface Vehicle

```

{
    public int getNoOfWheels();
}

class DefaultVehicle implements Vehicle
{
    public int getNoOfWheels()
    {
        return 2;
    }
}

```

Every class provide an interface by default Public & Static so directly create object

```

class Bus implements Vehicle
{
    public int getNoOfWheels()
    {
        return 6;
    }
}

class Test
{
    public static void main(String[] args)
    {
        Vehicle d = new Vehicle.DefaultVehicle();
        System.out.println(d.getNoOfWheels()); // 2

        Bus b = new Bus();
        System.out.println(b.getNoOfWheels()); // 6
    }
}

```

In the above example DefaultVehicle is Default implementation of Vehicle interface. whereas Bus is customized implementation of Vehicle interface.

### Notes

The class which is declared inside interface is always public static whether we are declaring or not hence we can create class object directly without having ~~the~~ outer interface type object.

### Conclusions

- (1) Among classes and interface we can declare anything inside
- Ex: Class A
  - { Class B
    - { Interface B
      - } static
- (2) The interface which is declared inside interface is always public & static whether we are declaring or not.
- (3) The class which is declared inside interface is always public and static whether we are declaring or not.
- (4) The interface which is declared inside a class is always static but need not be public.

- The process of designing applications in such a way that to provide support for various languages and various countries is called internationalization.
- If we are getting request from India then we should provide response to India people understandable forms.
- If we are getting request from US then we should provide response to US People understandable forms.
- We can implement internationalization concept by using the following 3 classes
  - (1) Locale
  - (2) NumberFormat
  - (3) DateFormat

### (1) Locale:

We can use Locale Object to represent a geographical region (or) Language.

→ Locale class present in java.util package, it is a final class and direct child of Object class.

→ It implements Serializable and Cloneable interfaces

(1) `Locale l = new Locale("string language");`  
creates a Locale object which represents the specified language.

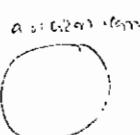
(2) `Locale l = new Locale("string language", "string country");`  
Language or Specified country.

Else `Locale l = new Locale("Pa", "IN");`  
creates a Locale object which represents Punjabi language of India country.

\* Locale class already defined some predefined Constants to represent Standard Locales.

`Locale.UK`  
`Locale.US`  
`Locale.ITALY`  
`Locale.ENGLISH`

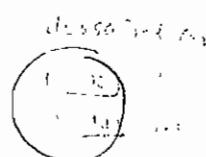
`setDisplayLanguage()`



723 723  
723 723  
723 723



723 723  
723 723  
723 723



## Important methods of Locale class

- ① Public static Locale getDefault();
- ② Public static void setDefault(Locale l);
- ③ Public String getCountry();
- ④ Public String getLanguage();
- ⑤ Public String getDisplayCountry();
- ⑥ Public String getDisplayLanguage();
- ⑦ Public String[] getISOLanguages();
- ⑧ Public String[] getISOCountries();
- ⑨ Public Locale[] getAvailableLocales();

Ex6

```
import java.util.*;  
class Localedemo  
{  
    public static void main(String[] args)  
    {  
        Locale l = Locale.getDefault();  
        System.out.println(l.getLanguage() + " --- " + l.getCountry());  
        System.out.println(l.getDisplayLanguage() + " --- " + l.getDisplayCountry());  
        Locale.setDefault(Locale.US);  
        System.out.println(Locale.getDefault().getDisplayCountry());  
        String[] s = locale.getISOLanguages();  
        for (String s1 : s)  
        {  
            System.out.println(s1);  
        }  
        String[] s1 = Locale.getISOCountries();  
        for (String s2 : s1)  
        {  
            System.out.println(s2);  
        }  
        Locale[] l1 = Locale.getAvailableLocales();  
        for (Locale l2 : l1)  
        {  
            System.out.println(l2.getDisplayCountry() + " --- " + l2.getDisplayLanguage());  
        }  
    }  
}
```

## ② NumberFormat

14-08-2014 [C8]

→ various countries follow various styles to represent numbers

Ex: 123456.789

IND → <sup>1</sup> 1,23,456.789

US → 123,456.789

ITALY → 123.456,789

- we can use NumberFormat object to represent a java number according to a locale specific string form.
- NumberFormat class present in java.text Package and it is an abstract class

NumberFormat nf = new NumberFormat(); X CE:

Getting NumberFormat object for default Locale

① Public static NumberFormat getInstance();

② Public static NumberFormat getCurrencyInstance();

③ Public static NumberFormat getPercentInstance();

④ Public static NumberFormat getNumberInstance();

Getting NumberFormat object for a particular Locale

The above methods are exactly same but as argument to these methods we have to provide the required Locale object.

Ex: ① Public static NumberFormat getNumberInstance(Locale l);

→ once if we get NumberFormat object we can call the following methods on that object

① Public String format(long e);

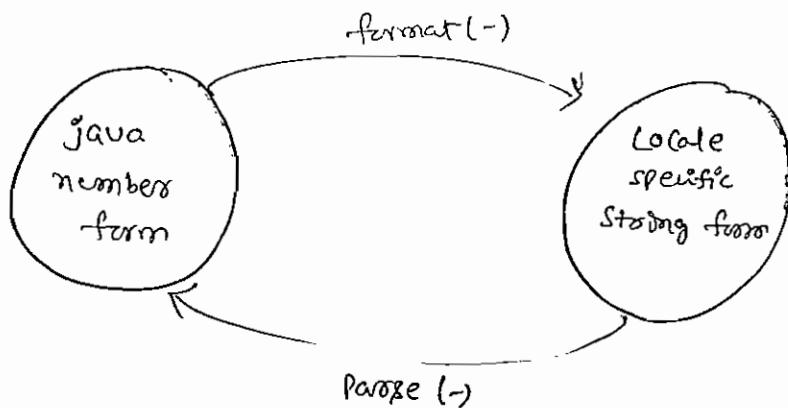
② Public String format(double d);

to convert java form to Locale specific string form

③ Public Number Parse(String s) throws ParseException

to convert locale specific string form to java number form

MF



P) Write a Program to represent a java number in ITALY specific numbers form

```
import java.text.*; // NumberFormat
import java.util.*; // Locale
class NumberFormatDemo2 {
    public static void main(String[] args) {
        double d = 123456.789;
        NumberFormat nf = NumberFormat.getNumberInstance(Locale.ITALY);
        System.out.println("ITALY form is: " + nf.format(d));
    }
}
```

O/P: ITALY form is: 123.456,789

P) Write a Program to represent a java number in INDIA, US & UK currency representations

```
import java.text.*;
import java.util.*;
class NumberFormatDemo1 {
    public static void main(String[] args) {
        double d = 123456.789;
        Locale India = new Locale("en", "IN");
        NumberFormat nf = NumberFormat.getNumberInstance(Locale.INDIA);
        System.out.println("India Notation is: " + nf.format(d));
        NumberFormat nf1 = NumberFormat.getNumberInstance(Locale.US);
        System.out.println("US Notation is: " + nf1.format(d));
        NumberFormat nf2 = NumberFormat.getNumberInstance(Locale.UK);
        System.out.println("UK Notation is: " + nf2.format(d));
    }
}
```

O/P →

India Notation is: ... INR 123,456.79  
US Notation is: ... \$ 123,456.79  
UK Notation is: ... £ 123,456.79

- Setting maximum and minimum integers and fraction digits

```

① public void setMaximumFractionDigits (int n)
② public void setMinimumFractionDigits (int n)
③ public void setMaximumIntegerDigits (int n)
④ public void setMinimumIntegerDigits (int n)

```

NumberFormat nf = NumberFormat.getInstance();

Case 1:

```

nf.setMaximumFractionDigits (2);
System.out.println (nf.format (123.4567)); // 123.46
System.out.println (nf.format (123.4)); // 123.4

```

Case 2:

```

nf.setMinimumFractionDigits (2);
System.out.println (nf.format (123.4567)); // 123.4567
System.out.println (nf.format (123.4)); // 123.40

```

Case 3:

```

nf.setMaximumIntegerDigits (3);
System.out.println (nf.format (123456.789)); // 123,456.789
System.out.println (nf.format (1.2345)); // 1,2345

```

Case 4:

```

nf.setMinimumIntegerDigits (3);
System.out.println (nf.format (123456.789)); // 123,456.789
System.out.println (nf.format (1.2345)); // 001.2345

```

### ③ DateFormat

- Various countries follow various styles to represent dates
- RND → 14 August, 2014
- US → 14 August 14, 2014
- We can use DateFormat object to represent current system date according to a Locale specific String form
- DateFormat class present in java.text package and it is an abstract class

DateFormat df = new DateFormat(); X c.e

Getting DateFormat object for default Locale

- ① Public static DateFormat getInstance()
- ② Public static DateFormat getDateInstance()
- ③ Public static DateFormat getDateInstance (int style)

## int style

DateFormat·0 FULL → [0] → Thursday, 14 August, 2014

DateFormat·1 LONG → [1] → 14 August, 2014

DateFormat·2 MEDIUM → [2] → 14 AUG, 2014

DateFormat·3 SHORT → [3] → 14/08/14

## Getting DateFormat object for a specific Locale

① public static DateFormat getDateInstance(int style, Locale l)

[15-08-2014]

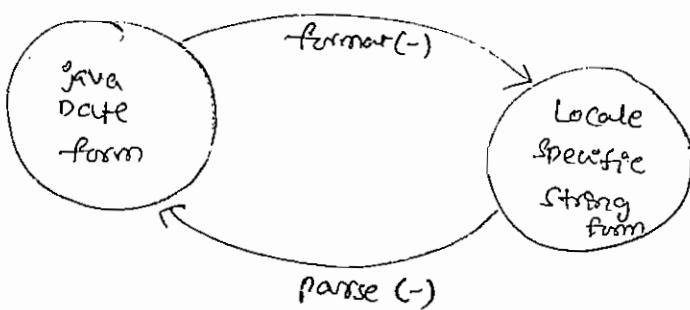
Once if you get DateFormat object we can call the following methods on that object.

① [public String format(Date d);]

To convert java Date form to locale specific String form

② [public Date parse(String s) throws ParseException]

To convert Locale specific String for to java Date form



Q) Write a program to represent current System date in US, UK and Italy Locale specific representations with full style?

import java.text.\*;  
import java.util.\*;

Class DateFormatDemo

```

public static void main(String[] args)
{
    Date d = new Date();
    DateFormat us = DateFormat.getDateInstance(0, Locale.US);
    DateFormat uk = DateFormat.getDateInstance(0, Locale.UK);
    DateFormat italy = DateFormat.getDateInstance(0, Locale.ITALY);

    System.out.println("US style is " + us.format(d));
    System.out.println("UK style is " + uk.format(d));
    System.out.println("Italy style is " + italy.format(d));
}
  
```

Q1PG

US style is : Friday, August 15, 2014

(17)

UK style is : Friday, 15 August 2014

ITALY style is : Venerdì 15 agosto 2014

Q1) Write a program to represent in all possible styles of India country

```
import java.text.*;  
import java.util.*;  
class DateFormatDemo1
```

```
{  
    public static void main (String[] args)  
    {  
        Date d = new Date();  
        Locale india = new Locale ("en", "IN");  
        System.out.println ("Full form is... " + DateFormat.getDateInstance (0, india).format (new Date ()));  
        System.out.println ("Long form is... " + DateFormat.getDateInstance (1, india).format (new Date ()));  
        System.out.println ("Medium form is... " + DateFormat.getDateInstance (2, india).format (new Date ()));  
        System.out.println ("Short form is... " + DateFormat.getDateInstance (3, india).format (new Date ()));  
    }  
}
```

O/P:

Full Form is : Friday, 15 August, 2014  
Long Form is : 15 August, 2014  
Medium Form is : 15 Aug, 2014  
Short Form is : 15/8/14.

getting DateFormat object to represent Date and Time

methods

- ① Public static DateFormat getDateInstance ()
- ② Public static DateFormat getDateInstance (int datestyle, int timestyle)
- ③ Public static DateFormat getDateInstance (int datestyle, int timestyle, Locale l)

Ex:

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

```
import java.text.*;
```

```
class Test
```

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

```
{  
    Locale india = new Locale ("en", "IN");  
    System.out.println ("India Date & Time style is : " + DateFormat.getDateInstance (0, 0, india).format (new Date ()));  
}
```

O/P:

India Date & Time style is : Friday, 2014 15 August, 2014 6:31:14 PM

1st

- Note ① Medium style is the default style for both date and time
- Note ② The <sup>only</sup> allowed values for Date and Time style is 0 to 3 (0, 1, 2, 3) If we are using any other number then we will get Runtime Exception saying Illegal Argument Exception

(P) Demo Program on Parse() method.

```

import java.util.*;
import java.text.*;
class ParseDemo
{
    public static void main (String [] args) throws ParseException
    {
        double d = 123456.7899;
        NumberFormat nf = NumberFormat.getCurrencyInstance ();
        String s = nf.format(d);
        System.out.println ("Local specific form : "+s);
        Number n = nf.parse(s);
        System.out.println ("Java Number form : "+n);
    }
}
O/P: Local specific form: Rs. 123,456.79
Java number form: 123456.79
  
```

## Javac

We can use Javac command to compile a single or group of java source files.

```
javac [Options] Test.java ↴
javac [Options] A.java B.java C.java ↴
javac [Options] *.*java ↴
↓
- Version
- d
- source
- cp / - classpath
- verbose
:
```

## Java

We can use Java command to run a single class file

```
java [Options] (Test) ↴ A B C ↴
↓                               Commandline arguments
- version
- D
- cp / - classpath
- eal / - esal / - asal / - da
```

Note we can compile Run only one any number of Source files at a time but we can class file at a time

## ClassPath

ClassPath describes the location where required .class files are available. Java compiler and Jvm will use class path to locate required .class file

⇒ By default Jvm will always searches in current working directory for the required .class file

If we set classpath explicitly then Jvm will search in our specified classpath location and Jvm won't search in current working directory.

⇒ We can set the classpath in the following ways

① By using Environment variable

Class Path :

This way of setting class path is permanent and will be preserved across system restarts.

whenever we are installing a permanent file in our system then this approach is recommended.

(2) At Command prompt level by using set command:

```
set classpath = c:\durga_classes
```

This way of setting classpath will be preserved only for particular Command prompt Once command prompt closes automatically classpath will be lost.

(3) At Command level by using -cp option:

```
java -cp c:\durga_classes Test
```

This way of setting class path will be preserved only for particular Command Once command execution completes automatically classpath will be lost.

#### Notes

Among ③ ways of setting class path, setting classpath at Command Level is recommended because dependent classes are varied from command to command.

→

Once we set the classpath we can run our program from any location.

→ once we set classpath JVM won't search in current working directory and it will always search on the specified classpath location only.

Ex: class Test

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

```
C:\durga_classes> javac Test.java
```

```
C:\durga_classes> java Test
```

```
O/P: classpathDemo
```

```
C:\> java Test
```

```
Res: NoClassDefFoundError : Test
```

```
C:\> java -cp c:\durga_classes Test
```

```
O/P: classpathDemo
```

D61> java -cp C:\durga-classes Test

O/P: classpath Demo

E1> java -cp C:\durga-classes Test

O/P: classpath Demo

C:\durga-classes> java -cp E: Test

R.E: NoClassDefFoundError: Test

C:\durga-classes> java -cp ; E: Test

O/P: class path Demo

Ex26

C:\  
F AStudent.class

```
public class AStudent
{
    public void m1()
    {
        System.out.println("I want job
                           immediately");
    }
}
```

D:\  
F ITIndustry.class

```
public class ITIndustry
{
    public static void main(String[] args)
    {
        AStudent a1 = new AStudent();
        a1.m1();
        System.out.println("you will get soon!!!");
    }
}
```

D61> javac AStudent.java

D61> javac ITIndustry.java

C.E: Can not find symbol  
symbol: class AStudent  
location: class ITIndustry

D61> javac -cp C: ITIndustry.java

R.E:

D31> java -cp C: ITIndustry

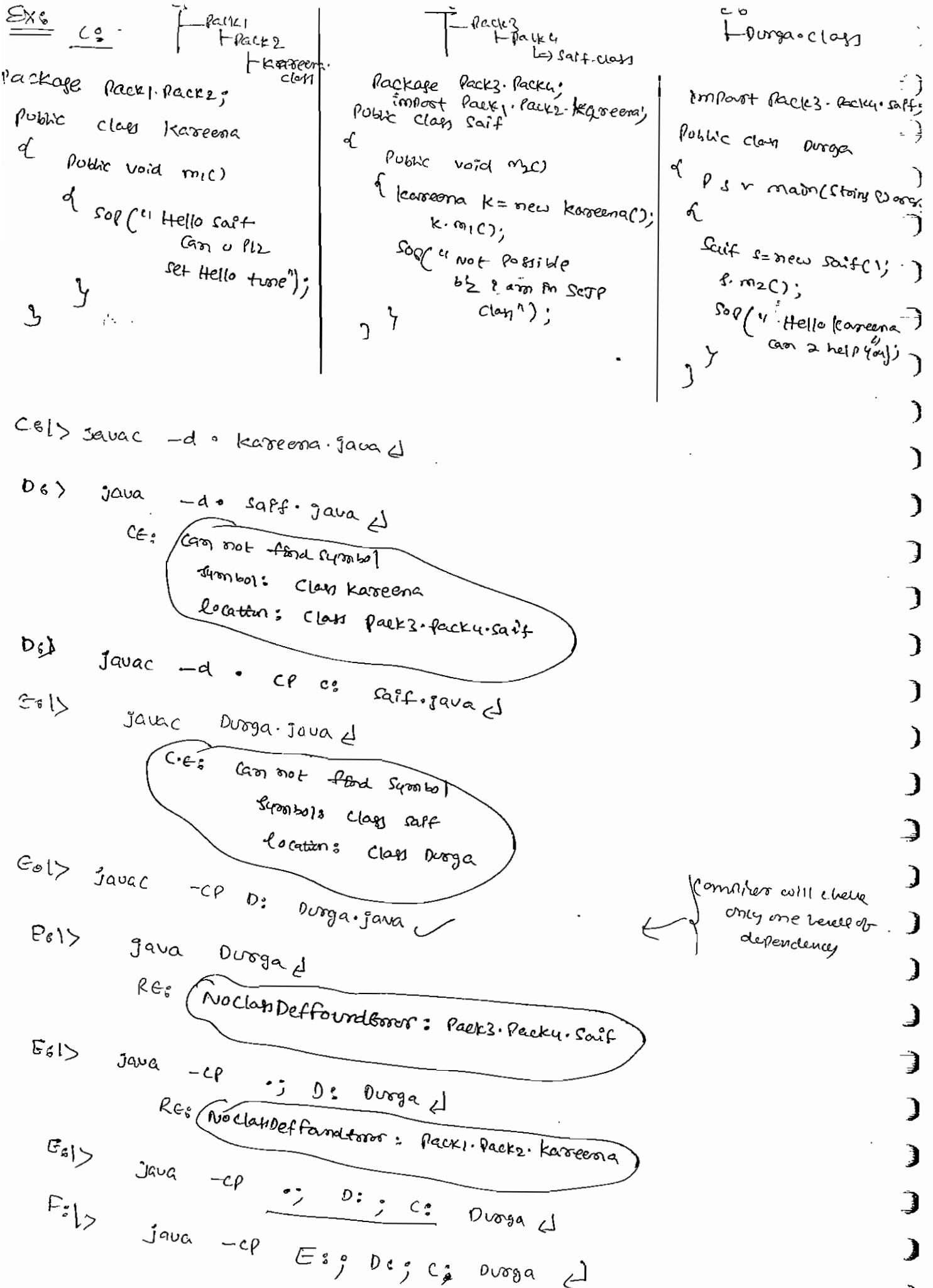
R.E: NoClassDefFoundError: AStudent

D61> java -cp C: ITIndustry

R.E: NoClassDefFoundError: ITIndustry

E61> java -cp ; C: ITIndustry

D:; C: ITIndustry



## Conclusion

- ① If any location created because of package statement that location should be resolved by using import statement and base package location we have to update in class path.
- ② Compiler will check only one level dependency whereas JVM will check all levels of dependency.
- ③ In classpath the order of locations is important and JVM will always consider from left to right until required match available

Ex: C:

```
public class Nagavalli
{
    P   S  V main(String[] args)
    {
        System.out.println("C: Nagavalli");
    }
}
```

D:

```
public class Nagavalli
{
    P   S  V main(String[] args)
    {
        System.out.println("D: Nagavalli");
    }
}
```

E:

```
public class Nagavalli
{
    P   S  V main(String[] args)
    {
        System.out.println("E: Nagavalli");
    }
}
```

Java -cp D:  
O/P: D:Nagavalli

Java -cp E;D;  
O/P: E:Nagavalli

Ses-2

Jar files if several dependent files are available then it is never recommended to set class file individually in class path we have to group all these class files into a single zip file and we have to make that zip file available in the classpath - this zip file is nothing but jar file.  
All third party SW plugging are by default available in the form of jar file only.

Eg: To develop a servlet all required dependent classes are available in `servlet-api.jar` we have to place this jar file in class path to compile a servlet program.

1. To run a JDBC program all dependent classes are available in `ojdbc14.jar` to run JDBC program we have to place this jar file in classpath.
2. To use log4j in our application dependent classes are available in `Log4j.jar` we have to place this jar file in the class path then only Log4j based application can run.

## Various Commands

### (1) To create a jar file : (ZIP file)

jar -cvf durgacalc.jar Test.class  
 ↘  
 Create verbose → named file

jar -cvf durgacalc.jar A.class B.class C.class

jar -cvf durgacalc.jar X.class

jar -cvf durgacalc.jar \*.\*

### (2) To extract a jar file (unzip file)

jar -xvf durgacalc.jar  
 ↘  
 Extract

### (3) To display table of contents:

jar -tf durgacalc.jar

Ex

## Service Provider's Role

(Demo program for jar file)

```
public class DurgaColorfulCode
{
  public static void add (int x, int y)
  {
    sum (x + y);
  }
  public static void multiply (int x, int y)
  {
    sum (x * x * y);
  }
}
```

javac Durgasoft DurgaColorfulCalc.java

jar -cvf durgacalc.jar DurgaColorfulCalc.class

(durgacalc.jar)

## Note:

To place .class file in class path just location is enough but to make jar file available in class path location is not enough Compulsory we have to include name or the jar file also

Client's Role

We downloaded jar file and we place in D: of client's machine

```

class Bakara
{
    public static void main(String[] args)
    {
        DurgaColorfulCalc.add(10, 20);
        DurgaColorfulCalc.multiply(10, 20);
    }
}

```

C:\durga-classes> javac Bakara.java X → CE;

C:\durga-classes> gavac -cp .\ Bakara.java X

C:\durga-classes> javac -cp D:\durgaCalc.jar Bakara.java

C:\durga-classes> java Bakara ↴  
Re: NoClassDefFoundError: DurgaColorfulCalc

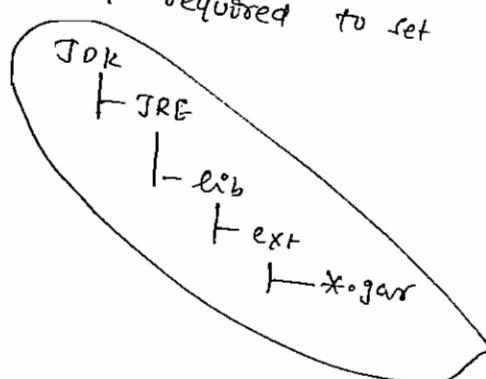
C:\durga-classes> java -cp .\ Bakara ↴  
Re: NoClassDefFoundError: Bakara

C:\durga-classes> java -cp .;\ D:\durgaCalc.jar Bakara ↴  
Re: NoClassDefFoundError: DurgaColorfulCalc

C:\durga-classes> java -cp .;\ D:\durgaCalc.jar Bakara ↴  
O/P: 300 --- 400

Shortcut way to place jar file in class path

If we place jar file in the following location then all classes and interfaces present in the jar file by default available to Java Compiler and JVM and we are not required to set classpath explicitly.



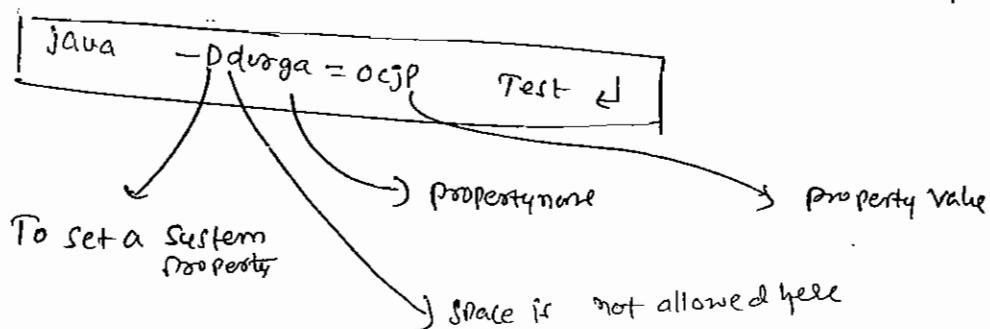
## System Properties

for every system some persistent information will be maintained in the form of system properties these include name of the OS, Java version, JVM vendor, user country etc...

Demo Program to print System properties:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Properties p = System.getProperties();
        p.list(System.out);
    }
}
```

→ we can set System Property explicitly from the command prompt by using -D option



The main advantage of setting System property is we can customize behaviour of Java Program.

Ex:-

```
class Test
{
    public static void main(String[] args)
    {
        String course = System.getProperty("course");
        if (course.equals("scjp"))
            System.out.println("SCJP Information");
        else
            System.out.println("Other course Information");
    }
}
```

Oracle SCJP

java -Dcourse=scjp Test  
Output SCJP Information  
java -Dcourse=sawd Test  
Output Other course Information

### ① jar vs war vs ear

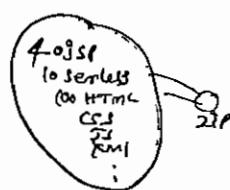
#### jar (Java Archive) :

It contains a group of .class files

#### war (Web Archive) :

A war file represents one web application which contains servlets, JSP's, ~~and~~ HTML pages, javascript files etc..

The main advantage of maintaining web application in the form of war file is project deployment, project delivery and project transportation will become easy.



#### ear (Enterprise Archive) :

An ear file represents one enterprise application which contains servlets, JSP's, EJB's, JMS components etc..

#### Notes

In general ear file represents a group of war files and jar files

### ② Web Application vs Enterprise Application

#### A Web Application

Can be developed by only web related technologies like servlets, JSP's, HTML, CSS files, Java script etc..

Ex:

Online Library management system.  
online shopping cart

An Java Enterprise Application can be developed by any technology from J2EE like servlets, JSP's, EJB's, JMS Components etc..

Ex: Banking Application

Telecom based Project etc...

#### Notes

J2EE or JEE compatible application for Enterprise Application.

### ③ Webserver vs Application Server

Webserver provides environment to run web applications

Webserver provides support for web related technologies like servlets, JSP's, HTML etc...

Ex: Tomcat

Application server provides environment to run enterprise applications

Application server can provide support for any technology from Java J2EE like servlets, JSP's, EJB's, JMS Components etc...

Ex: Weblogic, Websphere, JBoss etc...

## Notes

① Every Application server contains inbuilt webserver to provide support for web related technologies.

② Tomcat compatible server is Application server

④ How to Create executable jar file?

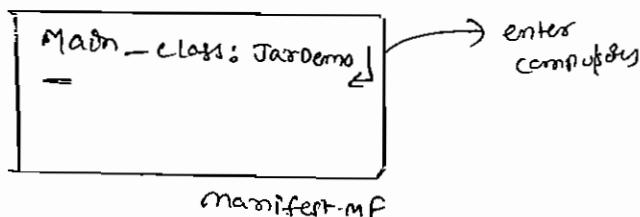
Program & JarDemo.java

```

import javax.awt.*;
import java.awt.event.*;
public class JarDemo
{
    public static void main(String[] args)
    {
        Frame f = new Frame();
        f.addWindowListener(new WindowAdapter()           → interface
        {
            public void windowClosing(WindowEvent e)
            {
                for (int i=1; i<=10; i++)
                    System.out.println("I am closing window: " + i);
                System.exit(0);
            }
        });
        f.add(new Label("I can create Executable jar file!!!"));
        f.setSize(500, 500);
        f.setVisible(true);
    }
}

```

manifest.MF



javac JarDemo.java



Jar -cvf demos.jar manifest.mf JarDemo.class JarDemo\$1.class



Java -jar demos.jar

Executing jar file

Even we can run .jar file by double clicking

Ques How many ways to run a java program?

We can run a java program in the following ways

① From command prompt we can run - class file with java command.

Eg: java jarDemo

② From Command prompt we can run .jar file with java command

Eg: java -jar demo.jar

③ By double clicking a .jar file

④ By double clicking a batch file

### Batch file

A batch file contains a sequence group of commands

Whenever we double click a batch file then all commands will be executed one by one in the sequence.

Eg:

```
java -cp c:\durga-classes JarDemo <
```

abc.bat



Ques Difference between Path and class Path?

#### Class Path:

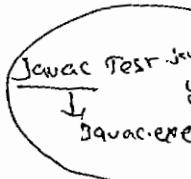
- 1) class Path describes the location where required class files are available
- 2) Java Compiler and Jvm will use class Path to locate required class files. If we are not setting class Path then our program may not compile and may not run.

#### Path:

Path describes the location where required binary executables are available

If we are not setting path then javac and java commands won't work.

Eg: Set Path = c:\Program files\java\jdk1.6.0\_11\bin



7

Difference b/w JDK, JRE and JVM?

JDK: (Java Development kit)

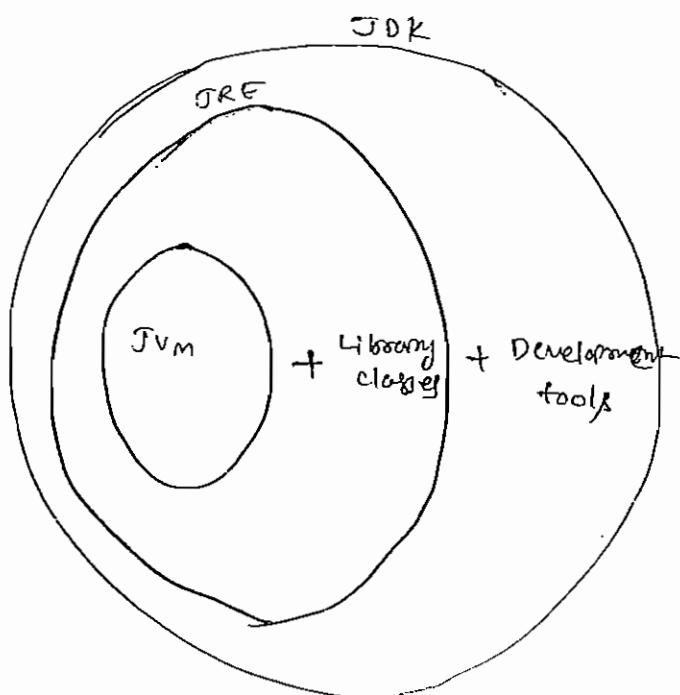
JDK provides environment to develop and run Java applications

JRE: (Java Runtime Environment)

JRE provides environment to run Java application

JVM: (Java Virtual Machine)

JVM is responsible to run Java programs line by line hence it is an interpreter.



JDK = JRE + Development tools

JRE = JVM + Library classes

JVM is the part of JRE whereas JRE is the part of JDK

Note :-

On the developer's machine we have to install JDK whereas on the client machine we have to install JRE.

8

Difference b/w java vs Javaw and javaws

Solu<sup>n</sup>:

We can use java command to run a Java class file where Sop's will be executed and corresponding output will be displayed to the console.

### javaw: (java without console output)

We can use javaw command to run a java class file where System.out.println's will be executed but the corresponding won't be displayed to the console.

- In general we can use javaw command to run GUI based applications

### Javaws: (Java web start utility)

We can use javaws to download a java application from the web and to start its execution

We can use javaws command as follows

javaws http://any url..

It downloads the application from the specified url and starts execution. The main advantage in this approach is every end user will get updated version and enhancement will become easy because of centralized control.

3

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

## Regular Expressions

If we want to represent a group of strings according to a particular pattern then we should go for regular expression.

Ex18 We can write a regular expression to represent all valid mobile numbers.

Ex19 We can write a ~~reg~~ regular expression to represent all mail id's

→ The main important application areas of regular expressions are

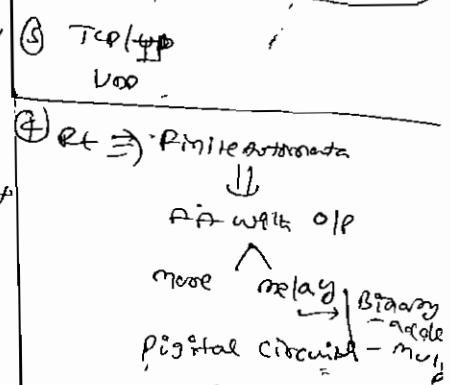
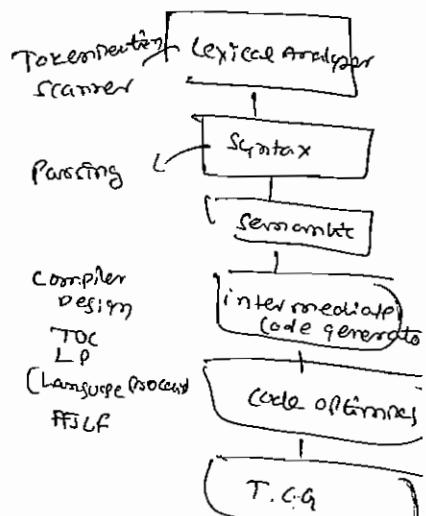
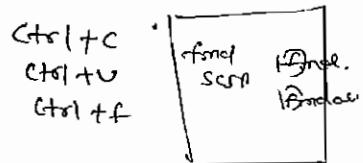
- (1) To develop validation frameworks
- (2) To develop pattern matching applications
- (3) To develop translators like Assemblers, Compilers  
Interpreters etc...
- (4) To develop digital circuits
- (5) To develop communication protocols like TCP / IP  
UDP etc...

Ctrl + f  
Windows  
Grep in UNIX

xxxxxx  
abc@gmail.com

Form Validation

name:	abc
Phone:	9876543210
DOB:	12/12/1990
sex:	Male
mail:	abc@gmail.com abc@cc.com
<input type="button" value="Submit"/>	



```

import java.util.regex.*;
class RegExDemo
{
    public static void main (String [] args)
    {
        int count=0;
        Pattern p = Pattern.compile ("ab");
        Matcher m = p.matcher ("abbabbbaab");
        while (m.find())
        {
            count++;
            System.out.println ("The total number of occurrences is : " + count);
        }
    }
}
0 --- 2 --- ab
3 --- 5 --- ab
The total number of occurrences is : 2
  
```

## Pattern

- A Pattern Object is a compiled version of regular expression i.e. it is a Java equivalent object of Pattern.
- We can create a Pattern object by using compile() method of Pattern class.

```
Public static Pattern compile(String re)
```

### Ex:

```
Pattern p = Pattern.compile("ab");
```

## Matcher

We can use Matcher object to check the given pattern in the target string.

- We can create a Matcher object by using matcher() method of Pattern class.

```
Public Matcher matcher(String target)
```

### Ex:

```
Matcher m = p.matcher("abbabbb");
```

### Important methods of Matcher class

#### ① boolean find()

It attempts to find next match and returns true if it is available.

#### ② int start()

Returns start index of the match.

#### ③ int end()

Returns end+1 index of the match.

#### ④ String group()

or returns the matched pattern.

## Note:

Pattern and Matcher classes present in java.util.regex package and introduced in 1.4 version.

(ab → pattern)

abbabbb

1 2 3

target string

## Character classes

$[abc]$   $\Rightarrow$  either 'a' or 'b' or 'c'

$[!abc]$   $\Rightarrow$  Except 'a' and 'b' and 'c'

$[a-z]$   $\Rightarrow$  Any lowercase alphabet symbol from a to z

$[A-Z]$   $\Rightarrow$  Any uppercase alphabet symbol from A to Z

$[a-zA-Z]$   $\Rightarrow$  Any alphabet symbol

$[0-9]$   $\Rightarrow$  Any digit from 0 to 9

$[0-9a-zA-Z]$   $\Rightarrow$  Any alphanumeric symbol

$[!0-9a-zA-Z]$   $\Rightarrow$  Except alphanumeric characters (special symbols)

## Ex:

Pattern  $P = \text{Pattern.compile}("x")$ ,

Matcher  $m = P.matcher("a3b#k@9z")$ ,

while ( $m.find()$ )

{

$sop(m.start() + "..." + m.group(1))$ ;

$x = [abc]$	$x = [!abc]$	$x = [a-z]$	$x = [0-9]$	$x = [a-zA-Z0-9]$	$x = [!a-zA-Z0-9]$
0 --- a					
2 --- b	1 --- #	0 --- a	1 --- 3	0 --- 9	3 --- #
	3 --- #	2 --- b	2 --- 6	1 --- 3	5 --- @
	4 --- k	4 --- k	4 --- 9	4 --- k	6 --- 9
	5 --- @	7 --- z	7 --- z	7 --- z	7 --- z
	6 --- 9				
	7 --- z				

## Predefined character classes

ls  $\Rightarrow$  Space character

IS  $\Rightarrow$  Except space character

ld  $\Rightarrow$  Any digit from 0 to 9

lD  $\Rightarrow$  Except digit, any character

lw  $\Rightarrow$  Any word character

lW  $\Rightarrow$  Except word character

[special characters]

\*  $\Rightarrow$  Any character

Ex:

Pattern p = Pattern.compile("a");

Matcher m = p.matcher("a**b k@9z");**

while (m.find())

{

    System.out.println(m.start() + "..." + m.group());

}

x == 118

3...  
0...a

x == 11S

0...a  
1...**7**  
2...**b**  
4...k  
5...@  
6...9  
7...z

x == 11q

1...7  
6...9

x == 11D

0...a  
2...b  
3...  
4...k  
5...@  
7...z

x == 11w

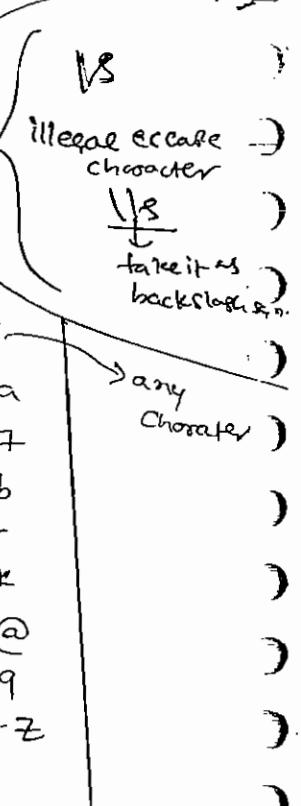
0...a  
1...7  
2...b  
4...k  
6...9  
7...z

x == 11W

3...  
5...@

x == e

0...a  
1...7  
2...b  
3...  
4...k  
5...@  
6...9  
7...z



## Quantifiers

We can use quantifiers to specify number of occurrences to match.

Quantifier → Quantity

a ⇒ Exactly one 'a'

a+ ⇒ Atleast one 'a'

a\* ⇒ Any no. of a's including zero number

a? ⇒ Atmost one 'a'

a? ⇒ 0/1  
0 or 1

atmost 1'a'  
0/1  
zero or one

abaabaaas  
↑  
→ 1st location or number  
in a's

Pattern p = Pattern.compile ("a");

Matcher m = p.matcher ("ababaaab");

while (m.find ())

{  
    System.out.println (m.start () + "..." + m.end());}

$x=a$	$x=at$	$x=a^*$	$x=a^?$
0---a	0---a	0---a	0---a
2---a	2---a	1---	1---
3---a	5---a	2---aa	2---a
5---a		4---	3---a
6---a		5---aaa	4---
7---a		8---	5---a
		9---	6---a
			7---a
			8---
			9---

↓  
9---

$x=a^*$
0---a
1---
2---a
3---a
4---
5---a
6---a
7---a
8---
9---

↓  
9---

$x=a^?$
0---a
1---
2---a
3---a
4---
5---a
6---a
7---a
8---
9---

↓  
9---

+  
At least one-  
One or more than one  
take as 1

and

ababaaab

reset end+1  
after b  
No character is the  
nothing  
so zero  
number

after 'b' nothing is there  
means zero number  
character

Regular expression we consider  
end+1 index

### Pattern class split()

We can use pattern class split() method to split the target string according to a particular pattern

Ex:

```
import java.util.regex.*;
class RegexDemo1
{
    public static void main (String[] args)
    {
        Pattern p = Pattern.compile ("\\s");
        String[] s = p.split ("Durga Software Solutions");
        for (String s1 : s)
            System.out.println (s1);
    }
}
```

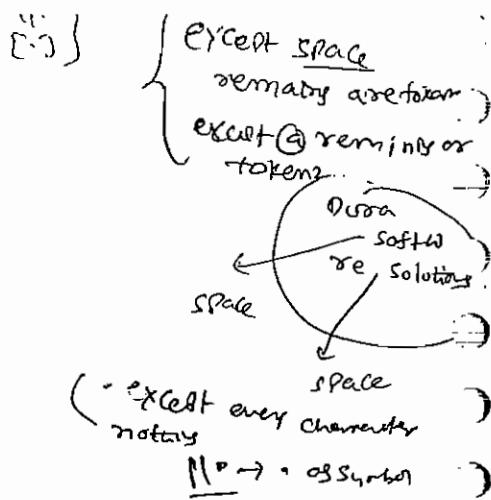
Output:  
Durga  
Software  
Solutions

## Ex 2g

```

Pattern p = Pattern.compile(" \w+ ");
String[] s = p.split("www.durgajobs.com");
for(String s1 : s)
    System.out.println(s1);
    
```

O/P: www  
durgajobs  
com



## String class Split() method

String class also contains `split()` method to split the target String according to a particular pattern.

### Ex 5

```

String s = "Durga software Solutions";
String[] s1 = s.split("\w+");
for(String s2 : s1)
    System.out.println(s2);
    
```

O/P : Durga  
software  
Solutions

## Note

Pattern class `split()` method can take target String as argument whereas String class `split()` method can take Pattern as argument.

## StringTokenizer

It is a specially designed class for tokenization activity.  
 StringTokenizer present in `java.util` package.

### Ex 19

```

StringTokenizer st = new StringTokenizer("Durga software Solutions");
while(st.hasMoreTokens())
{
    System.out.println(st.nextToken());
}
    
```

O/P : Durga  
software  
Solutions

Note: The default regular expression for StringTokenizer is space.

E92 6

```

StringTokenizer st = new StringTokenizer("19-09-2014", "-");
while (st.hasMoreTokens())
{
    System.out.println(st.nextToken());
}

```

O/p:

19
09
2014

With respect to which pattern

Requires explicit pattern definition

(Q) write a Regular expression to represent all valid 10 digit mobile numbers  
Rules

Rules 117

- 1) Every number should contain exactly 10 digits
  - 2) The first digit should be 7 (or) 8 (or) 9

(a digit)

[7-9] [0-9] {9}

↳ 9 times

10-digit (or)    11-digit

° ? [7-9] [0-9] {9}

10-digit | 11-digit | 12-digit

(0[9])? [7-9] [0-9] {9}

$$\begin{array}{l} 11 \rightarrow 0 \\ 12 \rightarrow 91 \end{array}$$

(e) write a regular expression to represent all valid magics?

$$[a-zA-Z0-9] [a-zA-Z0-9]^{*}$$

Page 1 of 1

$$-20-9] \{ a-2A \} 7_{a-9} + 7^k \{ a-1 \} = 2$$

$$[9-2A-2]^+$$

d123-xyz-k@gmail.com  
Not case sensitive

✓  
Yahoo.  
File

TV8 - net  
4shared

4 shared  
+

6 + at least one character

only gmail id's:

[a-zA-Z0-9] [a-zA-Z0-9\_]\* @ gmail.com

i) write a Regular expression to represent all Java Language Identifiers?

Rules:

(1) allowed characters are

a to z

A to Z

0 to 9

#

\$

(2) length of each identifier should be at least 2

(3) The first character should be lowercase alphabet symbol from a to z

(4) Second character should be a digit divisible by 3 (0, 3, 6, 9)

[a-k] [0369] [a-zA-Z0-9 #\$\_]\*

{ [a-bc] or [a-cd]

ii) write a program to check whether the given number is a valid mobile number (or) not? REG

```
import java.util.regex.*;
```

```
Class RegexDemo
```

```
{ P = Pattern.compile ("^(0|91)[7-9][0-9]{9}$");
```

```
Pattern P = Pattern.compile ("^(0|91)[7-9][0-9]{9}$");
```

```
Matcher m = P.matcher (args[0]);
```

```
if (m.find() & m.group().equals(args[0]))
```

```
{ System.out.println ("Valid mobile number"); }
```

```
else { System.out.println ("Invalid mobile number"); }
```

→ 98480 22338  
→ 098480 22338  
→ 919293949596  
X9292929293929 →  
↑ Invalid mobile no.  
for validation purpose.

→ It will pass if p from command prompt.  
at most one match  
only one match so no while loop just take if

→ match is there & total thing matched or not check.

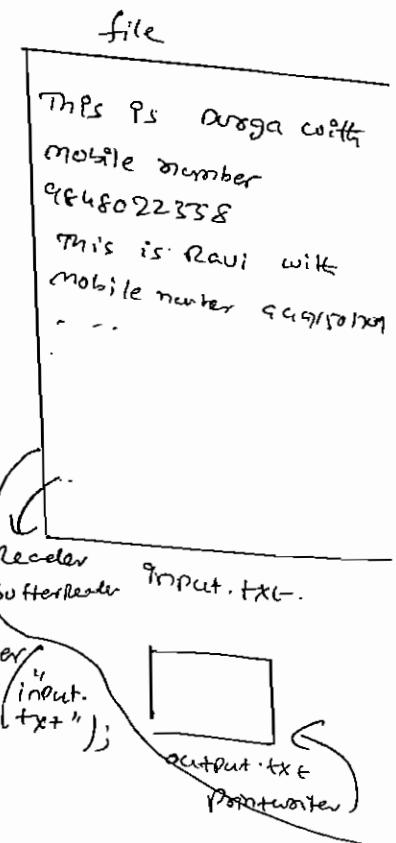
→ if which thing matched which provided thing is equal or not

- Q) write a program to check whether the given mail id is valid or not?  
 In the above program we have to replace mobile number regular expression with mail id regular expression

- Q) write a program to read all mobile numbers present in the given input file where mobile numbers are mixed with normal text data

```

import java.io.*;
import java.util.regex.*;
class MobileExtractor {
    public static void main(String[] args) throws Exception {
        PrintWriter out = new PrintWriter("output.txt");
        Pattern p = Pattern.compile("(0[1-9]{1})?([7-9]{1}[0-9]{9})");
        BufferedReader br = new BufferedReader(new FileReader("input.txt"));
        String line = br.readLine();
        while (line != null) {
            Matcher m = p.matcher(line);
            while (m.find())
                out.println(m.group(0));
            line = br.readLine();
        }
        out.flush();
    }
}
  
```



- Q) write a program to extract all mail id's present in the given input file where mail id's are mixed with normal text data

In the above program we have to replace mobile number regular expression with mail id regular expression

- Q) write a program to display all .txt file names present in C:/durga\_classes

$[a-zA-Z0-9-\cdot\$\cdot]^{+} \cdot [a-zA-Z0-9] \cdot txt$

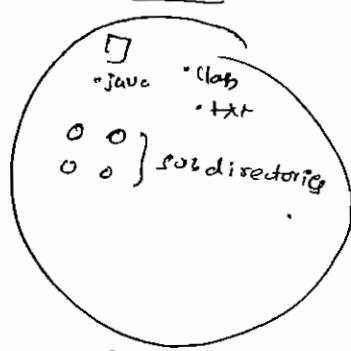
[ ] why

special meaning is there  
with out [ ] it take  
any character

abc.txt  
123abc.txt  
X abc.txt.bak  
not consider

abc\_def.txt  
abc\_def.txt  
abc\_def.txt  
+ any number of time

### folder



```

Ex
import java.io.*;
import java.util.regex.*;
class fileNameExtractor
{
    public static void main(String[] args) throws Exception
    {
        int count=0;
        Pattern p = Pattern.compile("[a-zA-Z0-9_]+[.]+[a-zA-Z]+");
        File f = new File("c://durga-classes");
        String[] s = f.list();
        for(String s1 : s)
        {
            Matcher m = p.matcher(s1);
            if(m.find() && m.group(1).equals(s1))
            {
                count++;
            }
        }
        System.out.println(count);
    }
}

```

Find all file names in  
 c://durga-classes  
 1) Create file object  
 2) display all file names within directory  
 3) total files matched or not  
 (abc.txt) all  
 either use group()  
 • java.util class  
 [ ] (Java class)  
 both

SCJP - 2<sup>nd</sup> edition

(18-08-2014)

## Enumeration (enum)

PPT

If we want to represent a group of named constants then we should go for enum.

Eg: enum month

```
{  
    JAN, FEB, MAR -- DEC;  
}
```

(; optional here)

enum Beer

```
{  
    KF, KO, RC, FO;  
}
```

;

(; ; is optional)

→ The main objective of enum is to define our own datatypes (Enumerated data types)\*

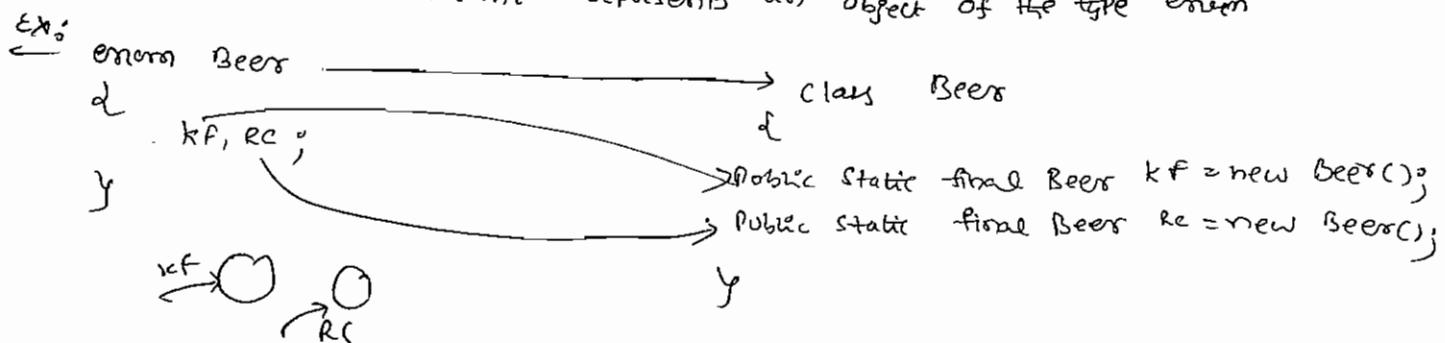
- enum concept introduced in 1.5 version of java
- when compared with old languages enum, Java enum is more powerful

{ source  
 Since → UML  
 -> class  
 & S6 }

### Internal implementation of enum

- Every enum will be converted into class
- Every enum constant is always public static final
- Every enum constant represents an object of the type enum

) it allows  
inheritance



### Enum declaration and usage

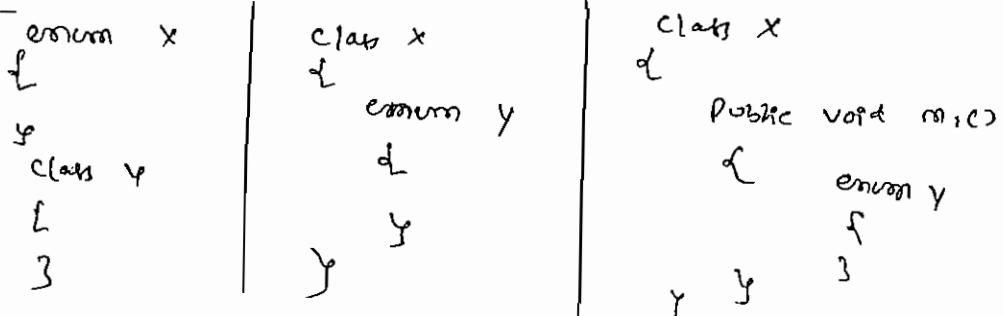
Every enum constant is public static final and hence we can access enum constants by using enum name

Eg:

```
enum Beers {  
    KF, KO, RC, FO;  
}  
class Test {  
    public static void main(String[] args) {  
        Beer b = Beers.KF;  
        System.out.println(b);  
    }  
}
```

- Inside enum `toString()` method internally implemented to return name of the constant.
- We can declare enum either outside the class (i) with in the class but not inside a method. If we are trying to declare inside a method then we will get compile time error

Ex:



C.G. { Enum types must not be local }

If we declare ~~enum~~ enum outside the class then the applicable modifiers are public, default, strictfp

If we declare enum with in the class then applicable modifiers are

public	default	+ private
strictfp		protected
		static

public  
default > (already fixed)  
final X  
abstract Y  
strictfp  
(Permitted)  
private  
protected  
static  
interface

### Enum vs Switch Statement

- Until 1.4 version the allowed argument types for the Switch Statement are byte, short, char, int
- but from 1.5 version onwards the corresponding wrapper classes and enum type also allowed
- from 1.7 onwards String type also allowed.

Switch (x) {  
 byte  
 char  
 short  
 int }

Switch (x)	1.4v	1.5v	1.7v
{	byte	Byte	
y	short	Short	
	char	Character	
	int	Integer	String
		+ enum	

hence from 1.5 version onwards we can pass enum type also as argument to switch statement.

Eg:

enum Beer

{ KF, KO, RC, FO;

}

Class Test

{ Public static void main (String [] args)

{ Beer b = Beer.RC;

switch (b)

case KF : sop ("it is childrens brand");  
break;

case KO : sop ("it is too legit");  
break;

case RC : sop ("it is not that much kick");  
break;

case FO : sop ("Buy one get one free");  
break;

default : sop ("other brands are not recommended").

O/P: it is not that much kick

If we are passing enum type as argument to switch statement then every case label should be valid enum constant otherwise we will get compilation error.

Eg:

switch (b)

{ case KF :

case RC :

case KO :

case FO :

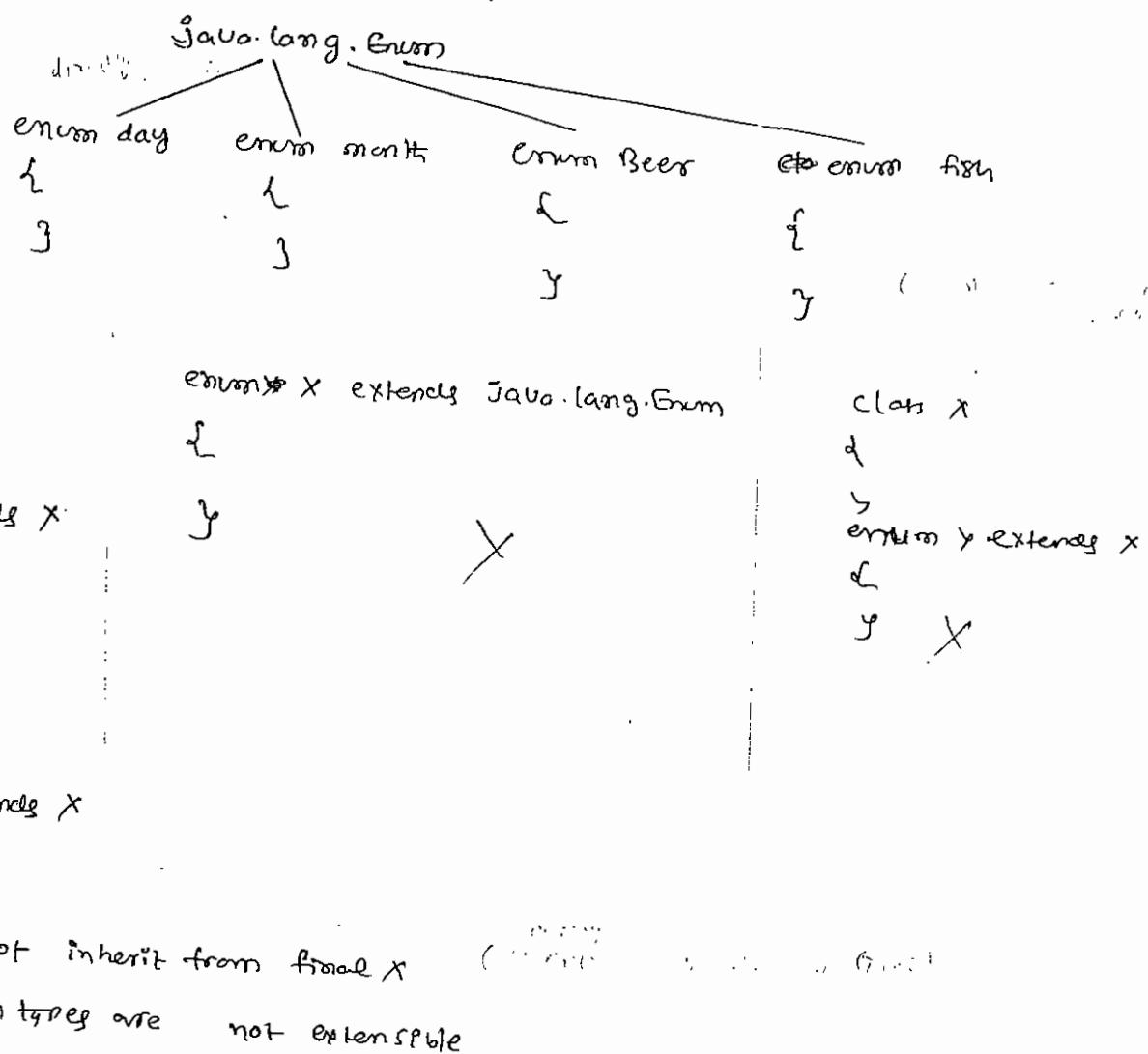
X case KALYANI :

}

↳ unqualified Enumeration constant name required.

## Enum vs Inheritance

- 1) every enum in java is the direct child class of `java.lang.Enum` class hence our enum can't extend any other enum
- 2) every enum is always final implicitly and hence we can't create child enum for our enum  
because of above reasons we can conclude inheritance concept not applicable for enum and hence we can't use `extends` keyword for enum.



• But an enum can implement any number of interfaces simultaneously

interface X

{ }

enum Y implements X

{ }

✓

## Java.lang.Enum

- Every enum in Java is the direct child class of java.lang.Enum hence this acts as base class for all Java enums.
- It is an abstract class and it is the direct child class of Object.
- It implements Serializable and Comparable interfaces.

### values()

We can use values() to list out all values present inside enum.

Every enum implicitly contains values() method.

Eg:

```
Beer [] b = Beer.values();
```

### Ordinal()

late constants

- Inside enum the order of constants is important and we can specify this order by using Ordinal value.
- We can find ordinal value of enum constant by using ordinal() method.

```
public final int ordinal();
```

- ordinal value is 0 (zero) based like array index.

### Program

```
enum Beer
{
    KF, KO, RC, FO;
}

class Test
{
    public static void main (String[] args)
    {
        Beer [] b = Beer.values();
        for (Beer b1 : b)
        {
            System.out.println(b1 + " --- " + b1.ordinal());
        }
    }
}
```

O/Ps	KF --- 0
	KO --- 1
	RC --- 2
	FO --- 3

### Speciality of Java enum

- In old languages enum we can take only constants but in Java enum in addition to constants we can take methods, normal variables, constructors etc... hence Java enum is more powerful than old languages.

→ we can declare main method inside enum and we can invoke enum class directly from command prompt.

Ex: enum Fish

```
{  
    STAR, GUPPY, GOLD;  
  
    Public static void main (String[] args)  
    {  
        System.out.println ("ENUM MAIN METHOD");  
    }  
}
```

Cmd  
javac Fish.java  
java Fish

Output: ENUM MAIN METHOD

→ In addition to constants if we are taking any extra member like a method then list of constants should be in the first line and should ends with semicolon (;)

enum Fish  
{  
 STAR, GUPPY;  
 Public void m1()  
 {  
 }  
}  
X  
mandatory

enum Fish  
{  
 STAR, GUPPY;  
 Public void m1()  
 {  
 }  
}  
X

enum Fish  
{  
 Public void m1();  
}  
X  
STAR, GUPPY;

→ Inside enum if we are taking any extra member like a method then first line should contain list of constants atleast ; (semicolon)

enum Fish  
{  
 Public void m1();  
}  
X

enum Fish  
{  
 ;  
 Public void m1();  
}  
X  
✓

→ but an empty enum is a valid java syntax.

(Q23)

```
enum fish {  
    }  
    }  
    ✓
```

```
enum fish {  
    ;  
}
```

[22-08-14]

### enum vs Constructors

enum class contains constructor and enum constructor will be executed separately for every enum constant at the time of enum class loading.

Ex:

```
enum Beer {  
    KF, KO, RC, FO;  
    Beer()  
    {  
        System.out.println("Constructor");  
    }  
}  
  
class Test  
{  
    public static void main(String[] args)  
    {  
        Beer b = Beer.RC; → ①  
        System.out.println("Hello");  
    }  
}  
  
Output:  
Constructor  
Constructor  
Constructor  
Constructor  
Hello
```



∴ every constant in a enum is object by default - it is public final static variable created at the time of class loading

If we comment line ① then the output is Hello.

- \* We can't create enum objects explicitly and hence we can't invoke enum constructor directly.

Eg: Beer b = new Beer();  
e.g.: enum types may not be instantiated

```
enum Beer {  
    KF, KO, RC, FO, b  
}  
for (var i : int i = 0; i < 5; i++)
```

## enum Beer

```
    KF(100), KO(75), RC(90), FO;  
    int price;  
    Beer(int price)  
    { this.price = price;  
    }  
    Beer()  
    { this.price = 65;  
    }  
    public int setPrice()  
    { return price;  
    }  
}  
  
class Test  
{  
    public static void main(String[] args)  
    {  
        Beer[] b = Beer.values();  
        for(Beer b1 : b)  
        {  
            System.out.println(b1 + " --- " + b1.getPrice());  
        }  
    }  
}
```

OP PG    KF --- 100  
          KO --- 75  
          RC --- 90  
          FO --- 65

## Notes

KF  $\Rightarrow$  Public Static final Beer    KF = new Beer();

KF(100)  $\Rightarrow$  Public Static final Beer    KF = new Beer(100);

## Enum Behavior

### Case 1

Every enum constant represents an object of the type enum and hence whatever the methods we can apply on normal java objects we can apply all those methods on enum constants also.

✓ Beer.KF.equals(Beer.RC)

✓ Beer.KF == Beer.RC

✓ Beer.KF.hashCode()  $\Rightarrow$  Beer.RC.hashCode()

X Beer.KF < Beer.RC

✓ Beer.KF.ordinal() < Beer.RC.ordinal()

Study: :-  
1. equals  
2. ==  
3. hashCode  
4. compareTo  
5. ordinal  
6. compareTo  
7. equals  
8. hashCode  
9. compareTo  
10. ordinal

### Case 2

If we want to use any class (or) interface from outside package directly then the required import is Normal import

If we want to access any static variable or method directly without class name then required import is static import.

Q25

Ex:

```
import static java.lang.Math.sqrt;  
import java.util.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        AL e = new AL();  
        e.sop(sqrt(4));  
    }  
}
```

→ with static import  
sop(Math.sqrt(4))

```
Package Pack1  
public enum Fish  
{  
    STAR, GUPPY;  
}
```

```
Package Pack2  
public class Test1  
{  
    public static void main(String[] args)  
    {  
        Fish f = Fish.Guppy;  
        sop(f);  
    }  
}
```

Required import is

import Pack1.Fish;  
(or)  
import Pack1.\*;

```
Package Pack3;  
public class Test2  
{  
    public static void main(String[] args)  
    {  
        sop(GUPPY);  
    }  
}
```

The required import is

import static Pack1.Fish.GUPPY;  
(or)  
import static Pack1.Fish.\*;

```
Package Pack4;  
public class Test3  
{  
    public static void main(String[] args)  
    {  
        Fish f = Fish.GUPPY;  
        sop(STAR);  
    }  
}
```

The required import is :

import Pack1.Fish; (or) import Pack1.\*;  
import static Fish.STAR;  
import static Pack1.Fish.\*;

Case 3 :-

### Ques 3: enum vs Enum vs Enumeration

① enum enum is a keyword in java which can be used to define a group of named constants

② Enum Enum is a class present in java.lang package. every Enum in java should be direct child class of this class (Enum class) hence Enum class acts as base class for all java ~~base~~ enums

③ Enumeration at is an interface present in java.util package we can use Enumeration to get objects one by one from collection

### Ques 4: enum color

```

enum color
{
    GREEN, RED, BLUE;
    public void info()
    {
        System.out.println("universal color");
    }
}

class Test
{
    public static void main(String[] args)
    {
        color[] c = color.values();
        for (color c1 : c)
        {
            c1.info();
        }
    }
}

```

O/P:

universal color  
universal color  
universal color

```

enum color
{
    GREEN, RED
    {
        public void info()
        {
            System.out.println("dangerous color");
        }
    },
    BLUE;
}

public void info()
{
    System.out.println("universal color");
}

class Test
{
    public static void main(String[] args)
    {
        color[] c = color.values();
        for (color c1 : c)
        {
            c1.info();
        }
    }
}

```

O/P:

universal color  
dangerous color  
universal color

# JVM ARCHITECTURE

Module-I

## Virtual Machine

### Types of virtual machines

1. Hardware Based virtual Machine
2. Application Based virtual MC

### - Basic Architecture of JVM

#### - Class Loader Subsystem

1. Loading
2. Linking
3. Initialization

#### - Types of Class Loaders

1. Bootstrap Class Loader
2. Extension Class Loader
3. Application Class Loader

### How class loader works

### What is the need of customized class loaders

### Pseudo code for customized class loader

## Module-II Various Memory Areas of JVM

1. Method Area
2. Heap Area
3. Stack Area
4. PC Registers
5. Native method Stack

### Program to display heap memory statistics

### How to set maximum and minimum heap size?

## Execution Engine

1. Interpreter
2. JIT Compiler

## Java Native Interface (JNI)

### Complete Architecture diagram of JVM

### Class File Structure.

## Virtual Machine

It is a software simulation of a machine which can perform operations like a Physical Machine.

There are ② types of virtual machines

① Hardware based / System based virtual machine.

② Application based / Process based virtual machine.

① Hardware based (or) System based virtual machine :

It provides several logical systems on the same computer with strong isolation from each other i.e. on one physical machine we are defining multiple logical ~~machines~~ machines.

The main advantage of hardware based virtual machines is hardware resource sharing and improves utilization of hardware resources.

Ex: KVM (Kernel based virtual m/c for Linux systems)

VMWare, Xen, Cloud computing etc....



② Application based (or) Process based virtual m/c.

These virtual machines acts as runtime engines to run

a particular programming language applications

Ex: 1) JVM (Java Virtual Machine) acts as runtime engine to run Java based applications

2) PVM (Parrot Virtual Machine) acts as runtime engine to run Perl based applications

3) CLR (Common Language Runtime) acts as runtime engine to run .Net based applications

CLR (Common Language Runtime) acts as runtime engine to run .Net based applications

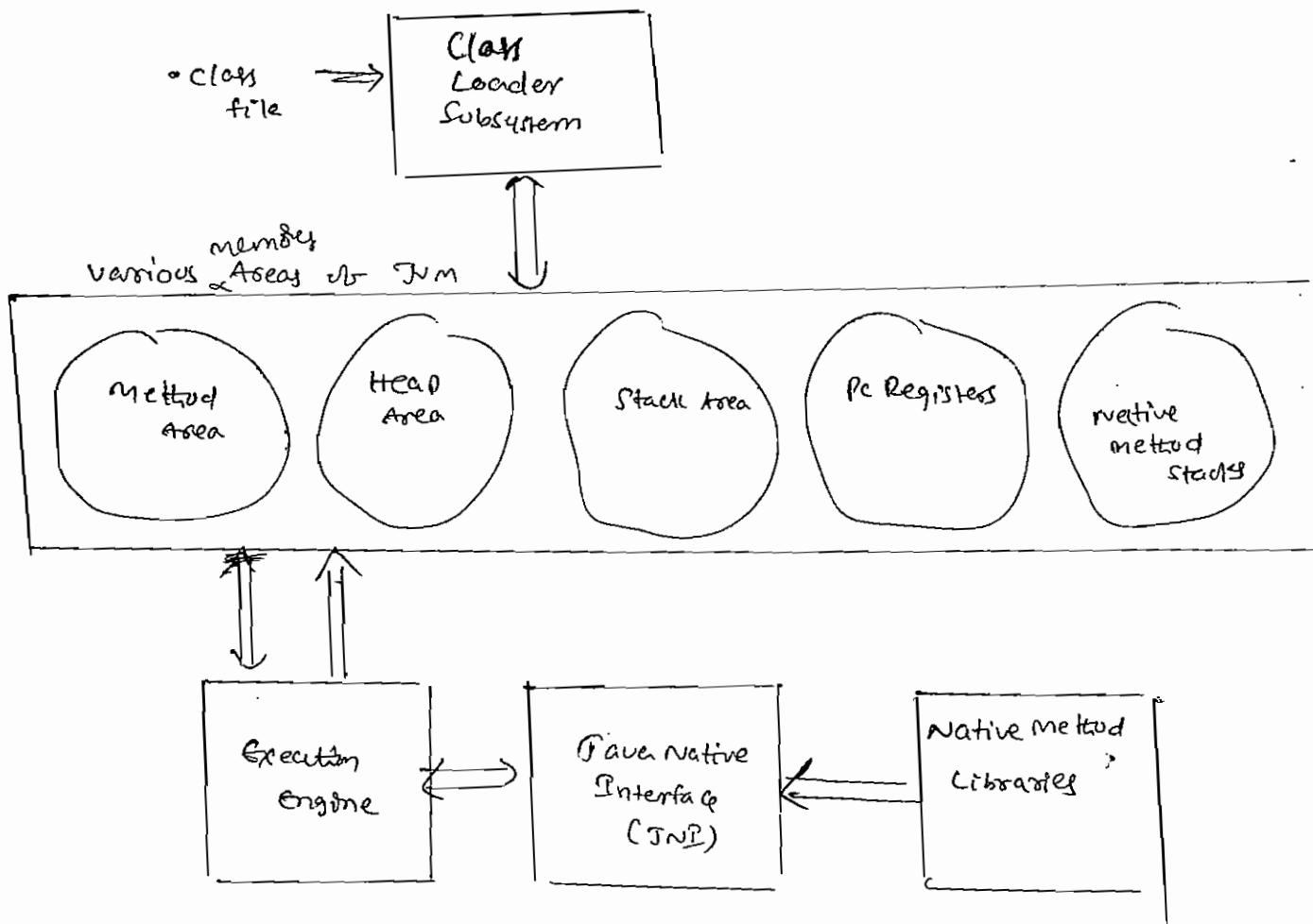
## JVM

JVM is the part of JRE and it is responsible to load and Run Java class files

Basic Architecture diagram of JVM

## Basic Architecture of JVM

927



## Class Loader Subsystem

Classloader subsystem is responsible for the following ③ activities

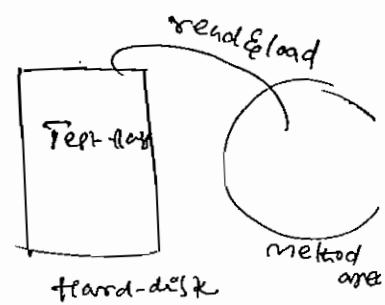
- ① Loading
- ② Linking
- ③ Initialization

### ① Loading:

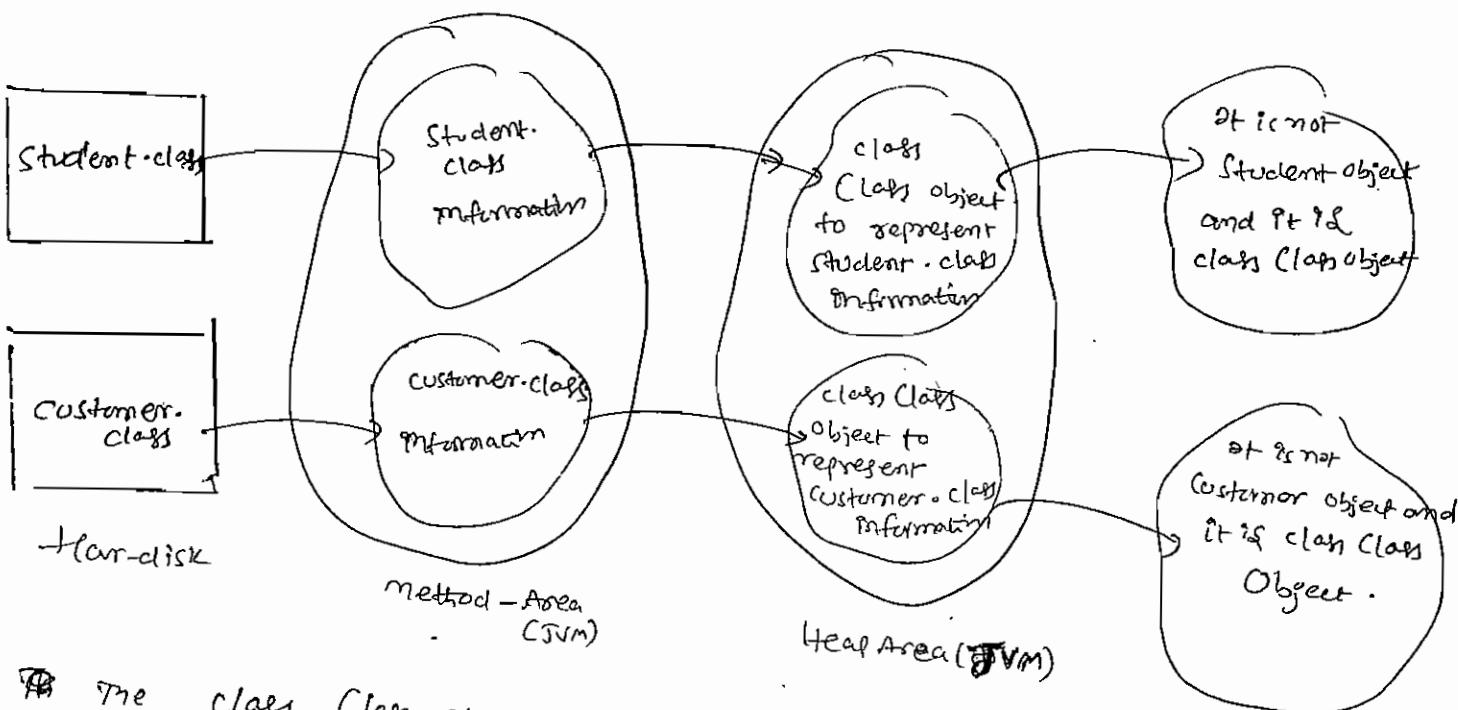
Loading means reading class files and store corresponding binary data in Method area.

for each class file JVM will store corresponding information in the method area.

- ① Fully qualified name of class
- ② fully qualified name of immediate Parent class
- ③ methods information
- ④ variables information
- ⑤ constructor information
- ⑥ modifiers information
- ⑦ Constant pool info etc..



After Loading .class file immediately JVM creates an object for that loaded class on the heap memory of java.lang.Class



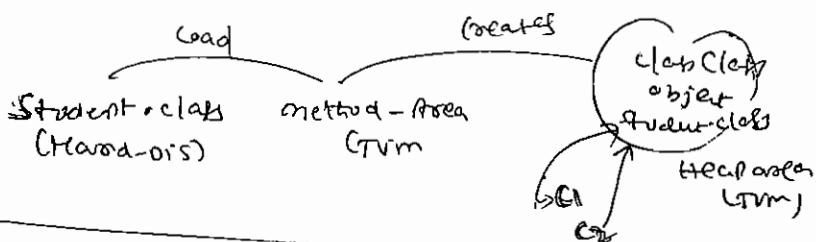
\* The `Class` object can be used by programmer to get class level information like methods information (or) variables information (or) constructors etc...

Ex:

```

import java.lang.reflect.*;
class Student
{
    public String getName()
    {
        return null;
    }
    public int getMarks()
    {
        return 10;
    }
}
class Test
{
    public static void main(String[] args) throws Exception
    {
        int count=0;
        Class c = Class.forName("student");
        Method[] m = c.getDeclaredMethods();
        for(Method m1:m)
        {
            count++;
            System.out.println(m1.getName());
        }
        System.out.println("The number of methods : "+count);
    }
}
  
```

Method() present in reflect API



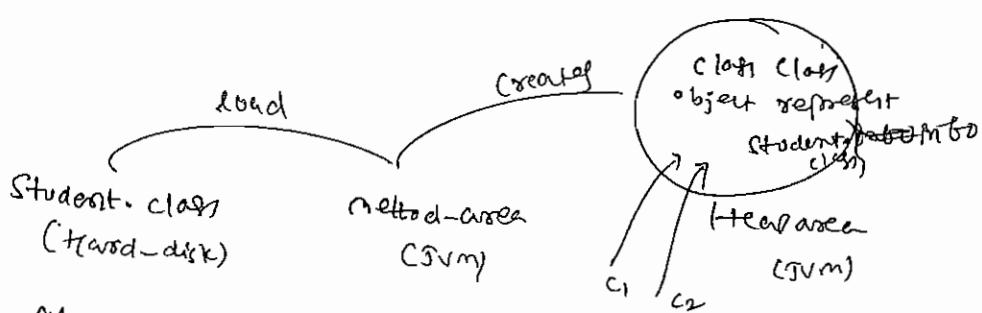
Student s1 = new Student;  
 Student s2 = new Student;  
 only one class object created  
 (23)

### Note 8

for every loaded type only one class object will be created even though we are using the class multiple times in our program

Ex:

```
import java.lang.reflect.*;
class Student {
    public String getName() {
        return null;
    }
    public int getMarks() {
        return 10;
    }
}
class Test {
    public void main(String[] args) {
        Student s1 = new Student();
        Class c1 = s1.getClass();
        Student s2 = new Student();
        Class c2 = s2.getClass();
        System.out.println(c1.hashCode()); 26117676
        System.out.println(c2.hashCode()); 26117676
        System.out.println(c1 == c2); true
    }
}
```



In the above program even though we are using student class multiple times only one class object got created.

## ② Linking :

Linking consists of ③ Activities

- ① Verify
- ② Prepare
- ③ Resolve

### ① Verify (or) verification:

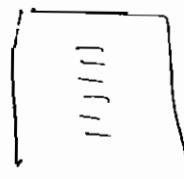
It is the process of ensuring that Binary representation of a class is structurally correct or not. JVM will check whether the class file is generated by valid compiler or not. i.e whether the class file is properly formatted (or) not.

• Internally Bytecode verifier is responsible for this activity.

• Bytecode verifier is the part of Class Loader subsystem.

• If verification fails then we will get Runtime exception saying [java.lang.VerifyError]

Bytecode verifier



Test.class

### ② Preparation:

In this ~~Preparation~~ phase static variables and assignments will allocate memory for class level Note: default values.

In initialization phase original values will be assigned to the static variables and here only default values will be assigned.

### ③ Resolution:

It is the process of replacing symbolic names from program with original memory references from method area.

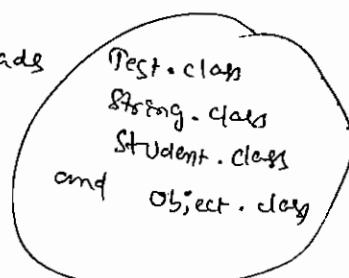
Eg:

= class Test

```

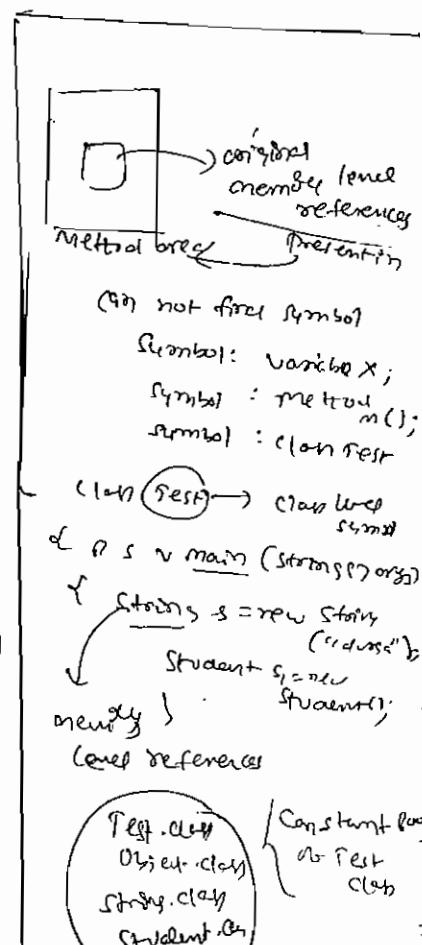
public static void main(String[] args)
{
    String s = new String("Hello");
    Student s1 = new Student();
}
  
```

for the above class Class Loader loads



the names of these classes are stored in Constant pool of Test class

In resolution phase these names are replaced with original memory level references from method area.



### ③ Initialization

- ↳ In this all static variables are assigned with original values and static blocks will be executed from Parent to child and from Top to bottom.

Class Loader Subsystem

↳ static  
↳ {  
↳ }  
↳ }

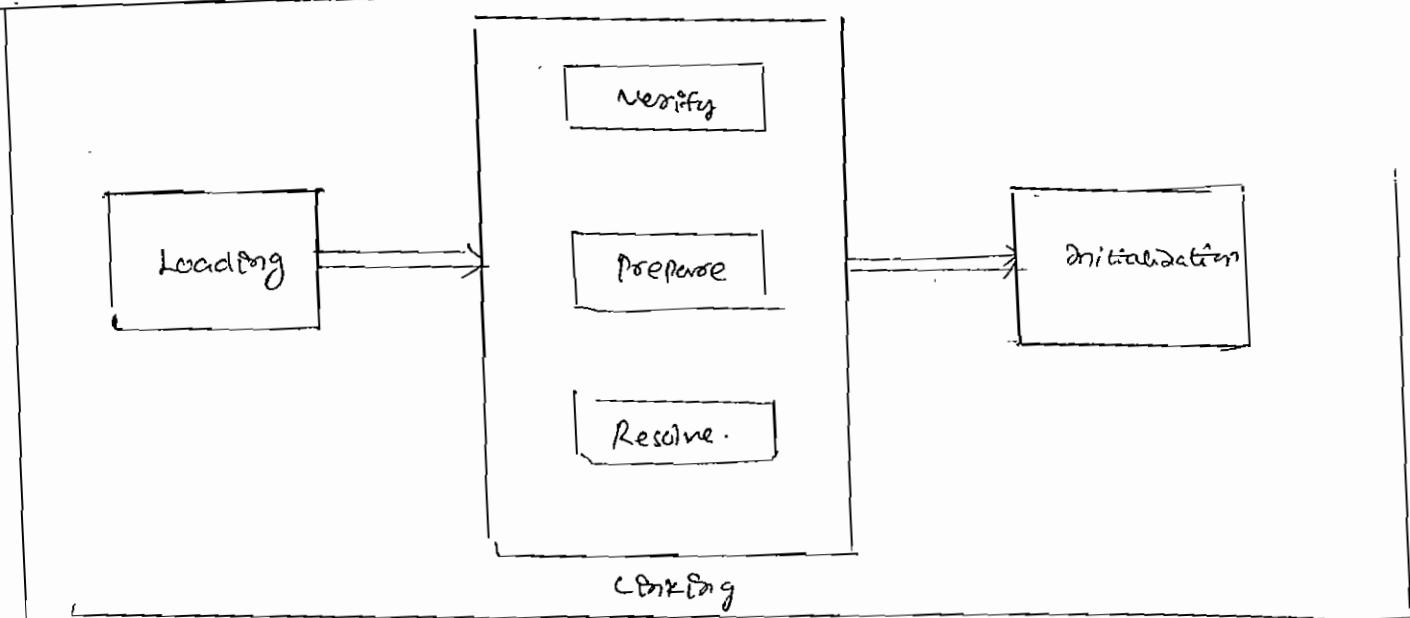


Fig: Class Loading Process

Note: While loading, linking and initialization if any error occurs then we will get runtime exception saying java.lang.LinkageError.

### TYPES OF CLASS LOADERS

Class Loader Subsystem contains the following ③ types of class loaders

Verify error is  
child or  
java.lang.LinkageError

- ① Bootstrap class loader (or) Primordial class Loader
- ② Extension class Loader
- ③ Application class Loader (or) System class Loader

#### ① Bootstrap class loader:

Bootstrap class loader is responsible to load core java API classes i.e. the classes present in rt.jar

(rt.jar  
core java  
programming)

JDK  
└─ JRE  
    └─ lib  
        └─ rt.jar

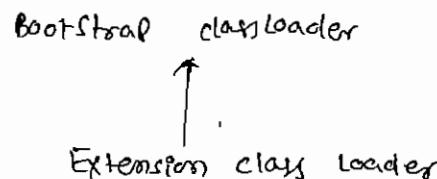
This location (JDK\jre\lib) is called Boot strap class path i.e. Boot strap class loader is responsible to load classes from Boot strap class path.

JDK  
└─ etc

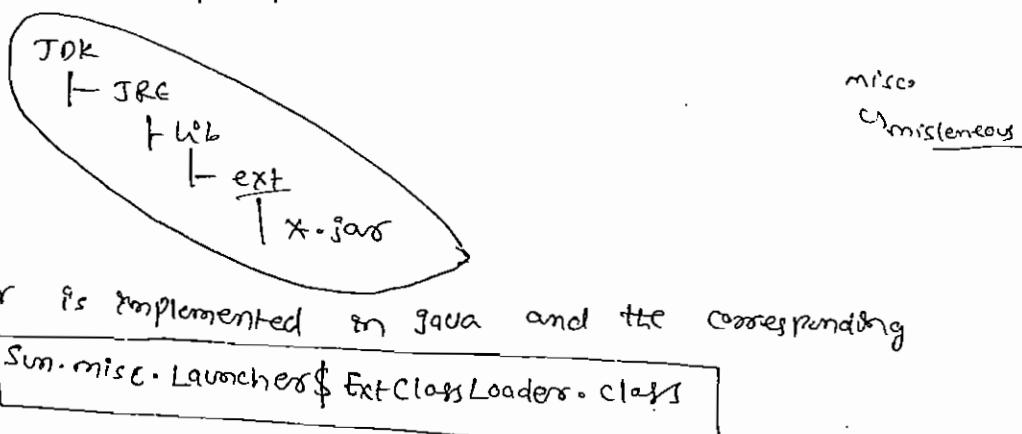
Bootstrap class loader is by default available with every JVM.  
It is implemented in native languages like C/C++ and not implemented in Java.

## 2) Extension class Loader

- 1) Extension class Loader is the child class of Bootstrap class Loader



- 2) Extension class Loader is responsible to load classes from extension-class path (JDK|JRE|lib|ext)



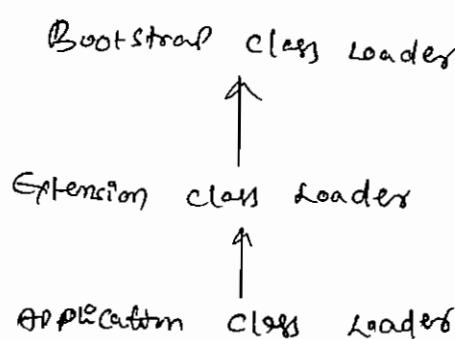
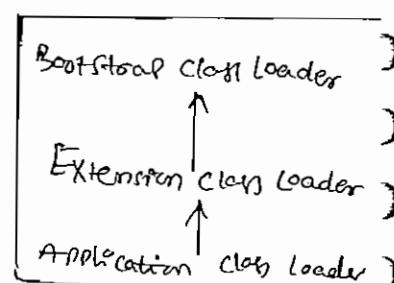
- 3) Extension class Loader is implemented in Java and the corresponding class file is Sun.misc.Launcher\$ExtClassLoader.class

## 3) Application class Loader (or) System class Loader

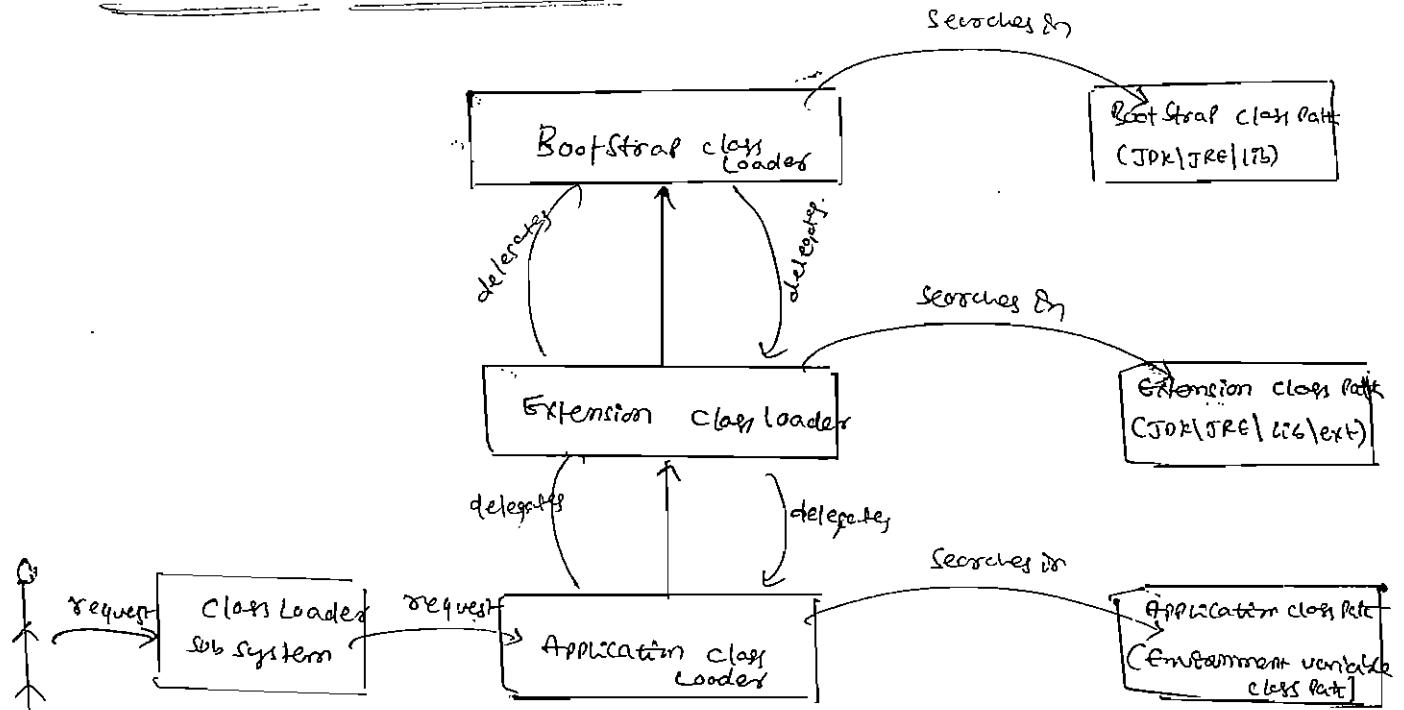
- 1) Application class Loader is the child class of Extension class Loader
- 2) This class Loader is responsible to load classes from Application class Path.

- 3) It internally uses environment variable class Path

- 4) Application class Loader is implemented in Java and the corresponding class file name is Sun.misc.Launcher\$AppClassLoader.class



## How Class Loader works



Class Loader follows Delegation Hierarchy principle.

Whenever JVM comes across a particular class first it will check whether the corresponding class file is already loaded or not if it is already loaded in method area then JVM will consider that loaded class.

If it is not loaded then JVM request class loader subsystem to load that particular class.

Then class loader subsystem handing over the request to Application class loader. Application class loader delegates the request to extension class loader which in turn delegates the request to bootstrap class loader.

Then Bootstrap class loader will search in Bootstrap class path if it is available then the corresponding class will be loaded by Bootstrap class loader. If it is not available then Bootstrap class loader delegates the request to Extension class loader.

Extension class loader will search in Extension class path if it is available then it will be loaded otherwise extension class loader delegates the request to Application class loader.

Application class loader will search in Application class path if it is available then it will be loaded otherwise we will get Runtime Exception saying

NoClassDefFoundError (or) ClassNotFoundException.

Ques

```

class Test
{
    public static void main (String[] args)
    {
        System.out.println(class.getClassLoader());
        System.out.println(Test.class.getClassLoader());
        System.out.println(Customer.class.getClassLoader());
    }
}

```

Ans: -> customer.java  
 Put this file inside src folder

Assume Customer.class present in Both extension and Application class path and Test.class present in only Application class path

for String.class

Bootstrap class loader from Bootstrap class path

for Test.class

Output is null (because it is not implemented in Java if it is not Java object)

Application class Loader from Application class path

(Cav0)

Output: Sun.misc.Launcher\$AppClassLoader@1912a56

↳ hashCode

for Customer.class

Extension class loader from extension class path

Output:

Sun.misc.Launcher\$ExtClassLoader@107a690

Note

- ① Bootstrap class Loader is not Java object hence we got null in the first case but Extension and Application class loader are Java objects hence we are getting corresponding output for the remaining ② sop's  
 [ Classname @ hashCode in hexadecimal form ]

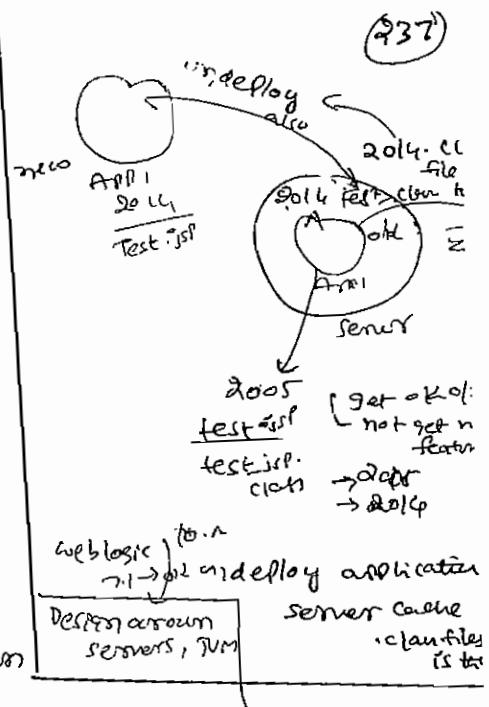
- ② Class Loader subsystem will give the highest priority for Application Bootstrap class Path and then Extension class Path followed by Application class Path.

## Need of customized class loader

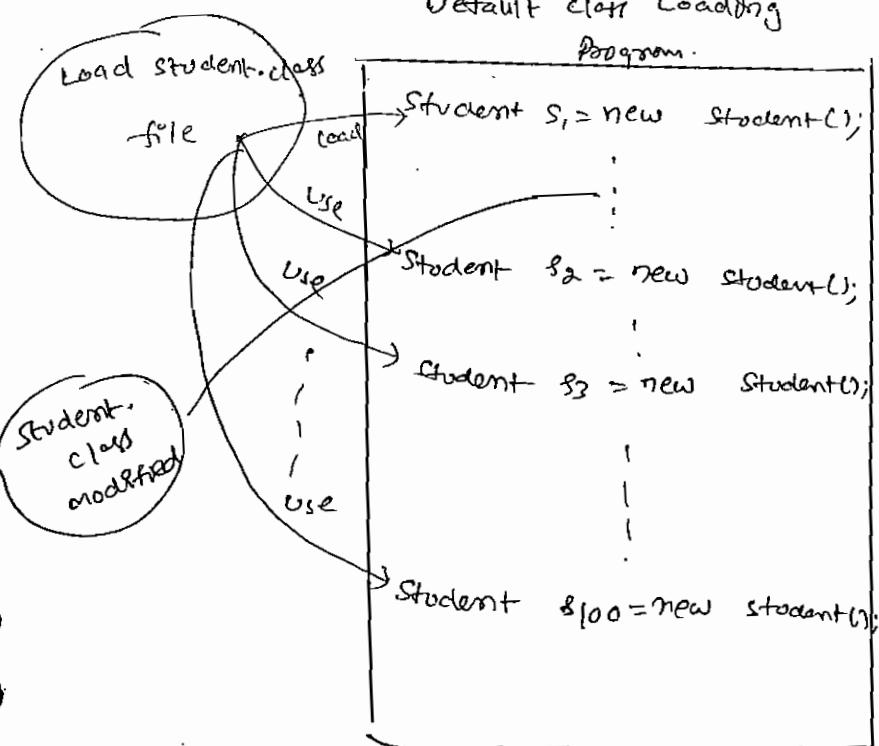
Default class loaders will load • class file only once even though we are using multiple times that class in our program after loading • class file if it is modified outside then default class loader won't load updated version of class file. (because • class file already available in method area)

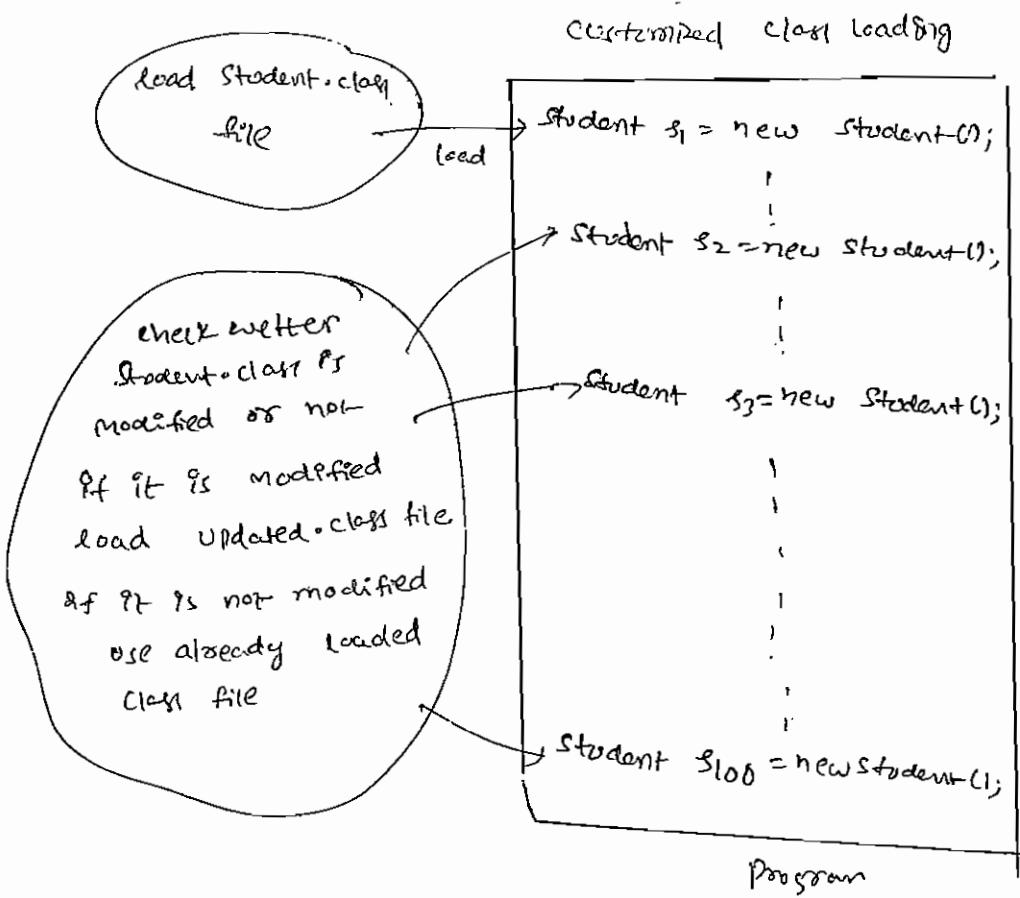
→ We can resolve this problem by defining our own customized class loader.

The main advantage of customized class loader is we can control class loading mechanism based on our requirement. for example so that updated version available to our program.



### Default class loading program.





### How to define customized class loader

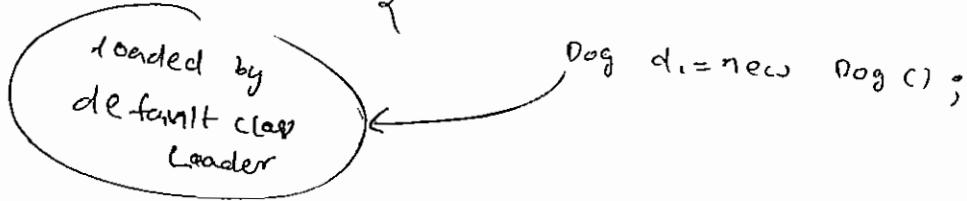
We can define our own customized class loaders by extending `java.lang.ClassLoader` class

Ex:

```

public class cost custClassLoader extends ClassLoader {
    public Class loadClass(String name) throws ClassNotFoundException {
        // check for updates & load updated .class file
        // and returns corresponding class
    }
}

class Client {
    public static void main(String[] args) {
        Dog d = new Dog();
    }
}
    
```



```

CustomClassLoader cl = new CustomClassLoader();
{
    cl.loadClass("Dog");
    cl.loadClass("Dog");
}
}

```

*loaded by  
Customized Class  
Loader*

Note While designing/developing web servers and Application servers usually we can go for customized class loaders to customize class loading mechanism.

Q) What is the need/use of ClassLoaders class?

We can use `java.lang.ClassLoader` class to define our own customized class loaders.

Every `ClassLoaders` in Java should be child class of `java.lang.ClassLoader` class either directly or indirectly. Hence ~~this~~ this class acts as base class for all customized class loaders.

## Module-2

Various memory areas inside JVM

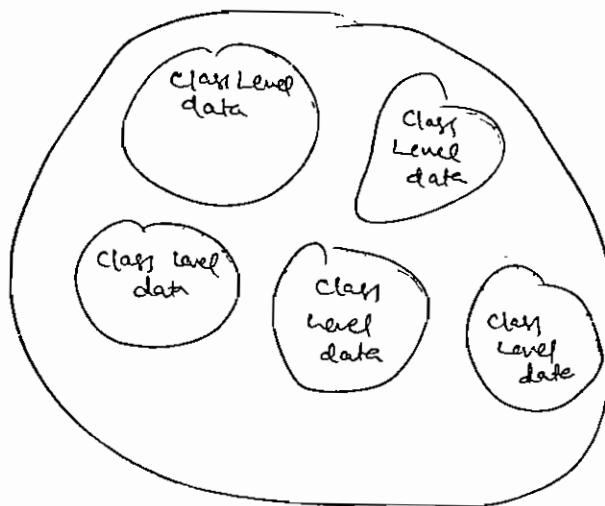
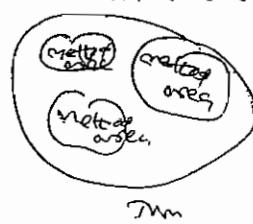
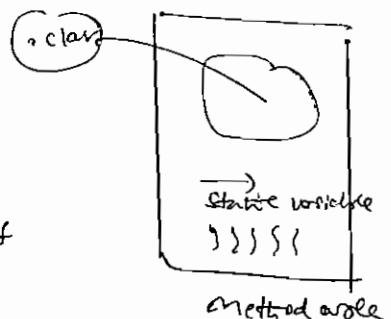
Whenever JVM loads and runs a Java program it needs memory to store several things like bytecode, objects, variables etc...

Total JVM memory organized into the following 5 categories

- (1) Method Area
- (2) Heap Area
- (3) Stack Memory
- (4) PC Registers
- (5) Native Method Stacks

## ① Method Area

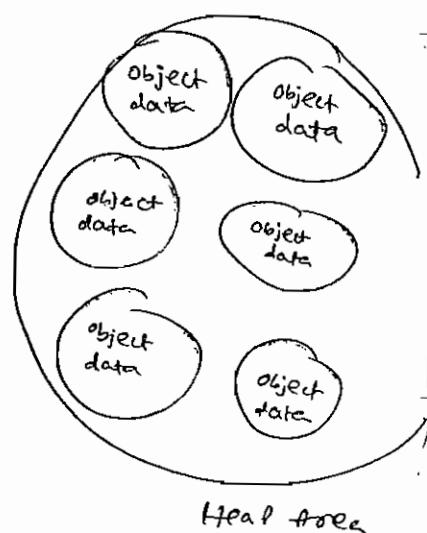
- (i) for every Jvm one method area will be available
- (ii) method area will be created at the time of jvm startup
- (iii) inside method area class level binary data including static variables will be stored
- (iv) constant pools of a class will be stored inside method area.



(v) Method Area can be accessed by multiple threads simultaneously

## ② Heap Area

- (i) for every Jvm one heap area is available
- (ii) heap area will be created at the time of jvm startup.
- (iii) objects and corresponding instance variables will be stored in the heap area
- (iv) every array in java is object only hence arrays also will be stored in the heap area.
- (v) heap area can be accessed by multiple threads and hence the data stored in the heap memory is not thread safe.
- (vi) heap area need not be continuous



11-08-2014

JVM Architecture  
Heap memory

## Program to display heap memory statistics

- A Java application can communicate with JVM by using Runtime object
- Runtime Class is a Singleton class we can create Runtime object by using `getRuntime()` method.

```
Runtime r = Runtime.getRuntime();
```

once we got Runtime object we can call the following methods on that object.

maxMemory():

it returns number of bytes to max memory allocated to the heap  
totalMemory():

it returns the numbers of bytes or total memory allocated to the heap  
freeMemory():

it returns number of bytes of free memory present in the heap.

### Program

```
class HeapDemo2 {
    public static void main (String [] args) {
        long mb = 1024 * 1024;
        Runtime r = Runtime.getRuntime();
        System.out.println("MaxMemory:" + r.maxMemory() / mb);
        System.out.println("Total Memory:" + r.totalMemory() / mb);
        System.out.println("Free Memory:" + r.freeMemory() / mb);

        System.out.println("Consumed Memory:" + (r.totalMemory() - r.freeMemory()) / mb);
    }
}
```

### O/P (in bytes)

MaxMemory : 257490944

Total memory : 16121856

FreeMemory : 15837944

Consumed memory : 283912

### O/P in (mb)

maxmemory : 245

TotalMemory : 15

FreeMemory : 15

Consumed memory : 0

245.5025

15.325

15.104260

0.270259 -

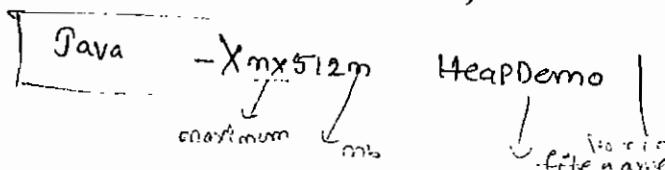
## \* How to set maximum and minimum heap size

Heap memory is a finite memory based on our requirement we can increase or decrease heap size. We can use the following flags for this requirement.

-Xmx

to set maximum heap size i.e. max memory

Eg:



↳ to set maximum heap size as 512 MB

O/P:

MaxMemory = 494

Total memory = 15

free memory = 15

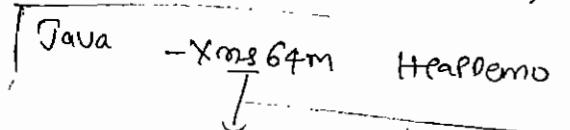
Consumed memory = 0

{ begin : 15 free memory }  
{ end : 15 free memory }

-Xms

to set ~~max~~ minimum heap size i.e. total memory

Eg:



to set minimum heap size as 64 MB

O/P:

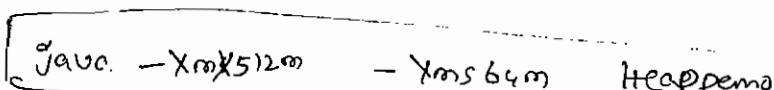
Maxmemory = 245

Total memory = 61

free memory = 61

Consumed memory = 0

-Xmx, -Xms



Max memory : 494

Total memory : 61

free memory : 61

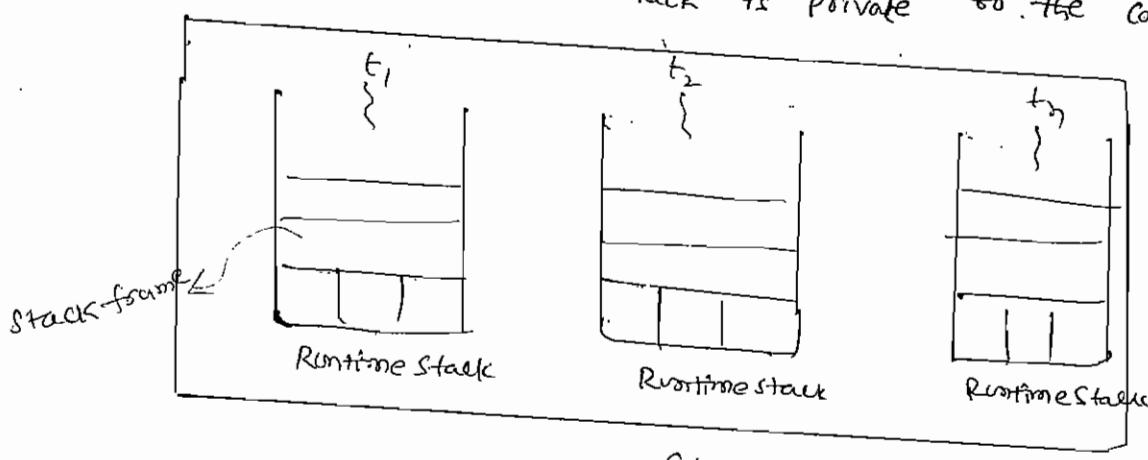
Consumed memory : 0

### (3) Stack Memory

for every thread JVM will create a runtime stack at the time of thread creation. Each and every method call performed by the thread and the corresponding local variables will be stored in the stack.

for every method call a separate entry will be added to the stack and the entry is called "Stack frame or activation Record".

- after completing the method call the corresponding entry will be removed from the stack.
- after completing all method calls the stack will become empty and that empty stack will be destroyed by the JVM just before terminating the thread.
- The data stored in the stack is private to the corresponding thread.



→ each stack frame containing 3 parts

a) local variable array :

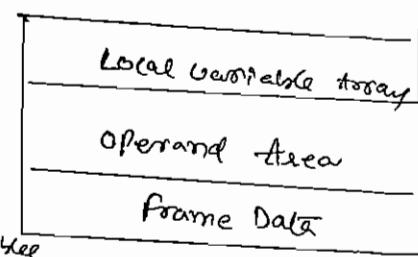
→ it contains all parameters and local variables of the method.

→ each slot for the array is of 4 bytes

occupied 1 slot for array & the reference (int, float, long, double)

→ values of long & double occupy 2 consecutive entries in the array.

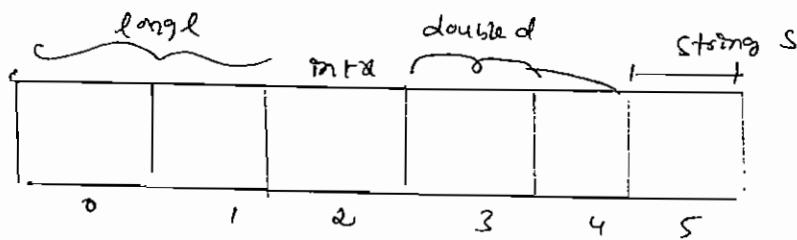
→ byte, short, char values will be converted to int type before storing & occupy one (1) slot.



Stack frame structure.

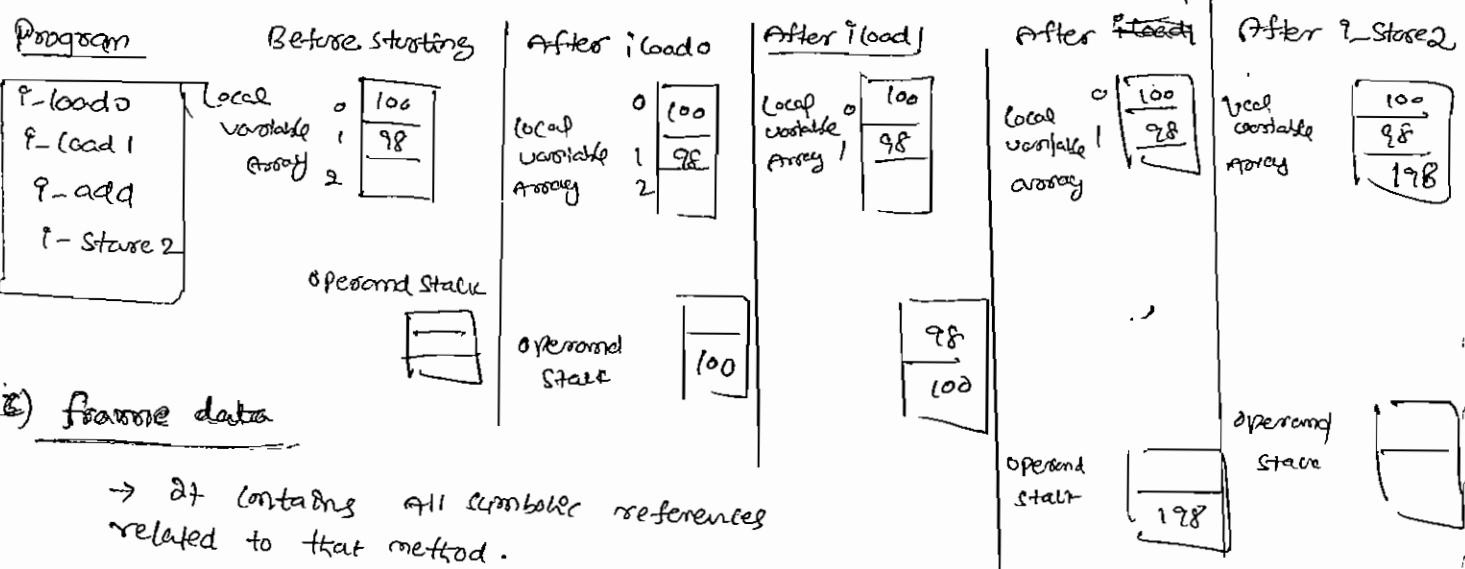
→ The way of storing boolean value is avoided from 0 to jvm because most of the JVM's follow one slot for boolean values.

Eg:-  
public static void m(long l, int x, double d, String s)  
{  
     $\Sigma$   
}



## 1 Operand Stack

- JVM uses operand stack as workspace
- Some instructions can push the values to the operand stack and some instructions perform required operations and some instructions store results etc...



## 2) Frame Data

- It containing all symbol references related to that method.
- It also containing a reference to exception table which provides corresponding catch block information in the case of exceptions.

## PC- Registers (Program Counter Registers)

12-08-14

- For every thread a separate PC register will be created at the time of thread creation.
- PC-registers contains address of current executing instruction
- Once instruction execution completed automatically PC-register will be incremented to hold address of next instruction

{ }  
My PC, Program

## Native method stacks

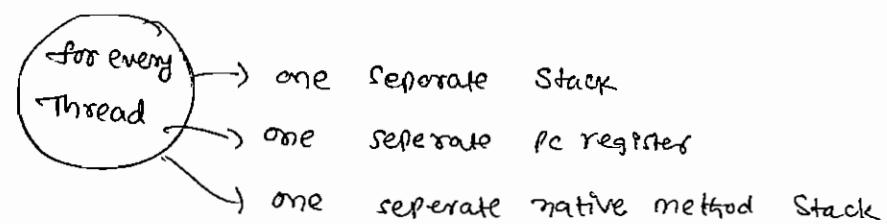
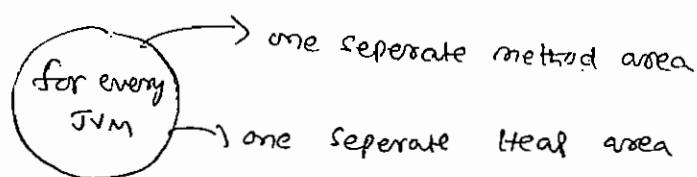
245

for every thread JVM will create a separate native method stack

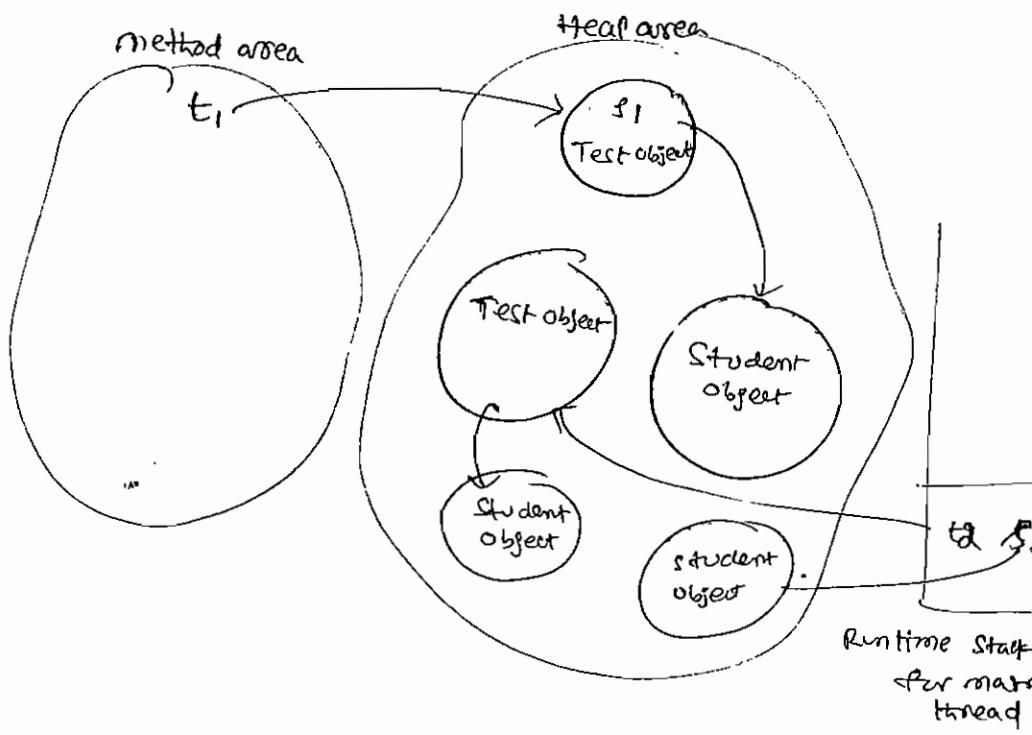
all native method calls invoked by the thread will be stored in the corresponding native method stack

Notes:

- ① Method Area, heap area and Stack area are considered as Important memory areas with respect to programmer
- ② Method Area and heap area are per JVM whereas Stack area, PC-registers and native method stacks are per thread.



- Static variables will be stored in method area
- Instance variables will be stored in heap area
- and local variables will be stored in stack area.



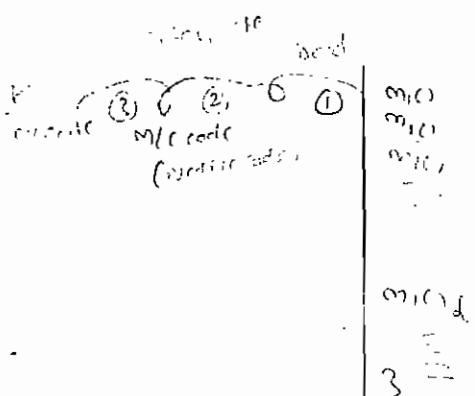
Class Test  
`2 Student s = new Student();`  
`Static Test t1 = new Test();`  
`Per main(String) arr`  
`{ Test t2 = new Test();`  
`Student s2 = new Student();`  
`}`  
`Stack of frame for main method`

## Execution Engine

- This is the central component of JVM
- Execution engine is responsible to execute Java class files
- Execution engine mainly contains 2 components for executing Java classes.
  - 1) Interpreter
  - 2) JIT compiler

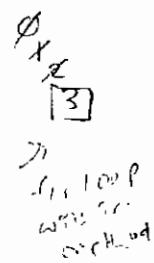
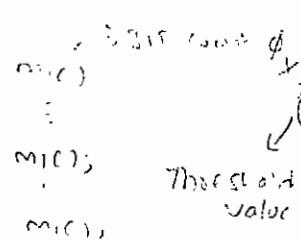
### ① Interpreter

- It is responsible to read bytecode and interpret it into machine code (native code) and execute that machine code line by line.
- The problem with interpreter is it interprets every time even same method invoked multiple times which effects performance of the system.
- To overcome this problem Sun people introduced JIT compilers in 1.4v



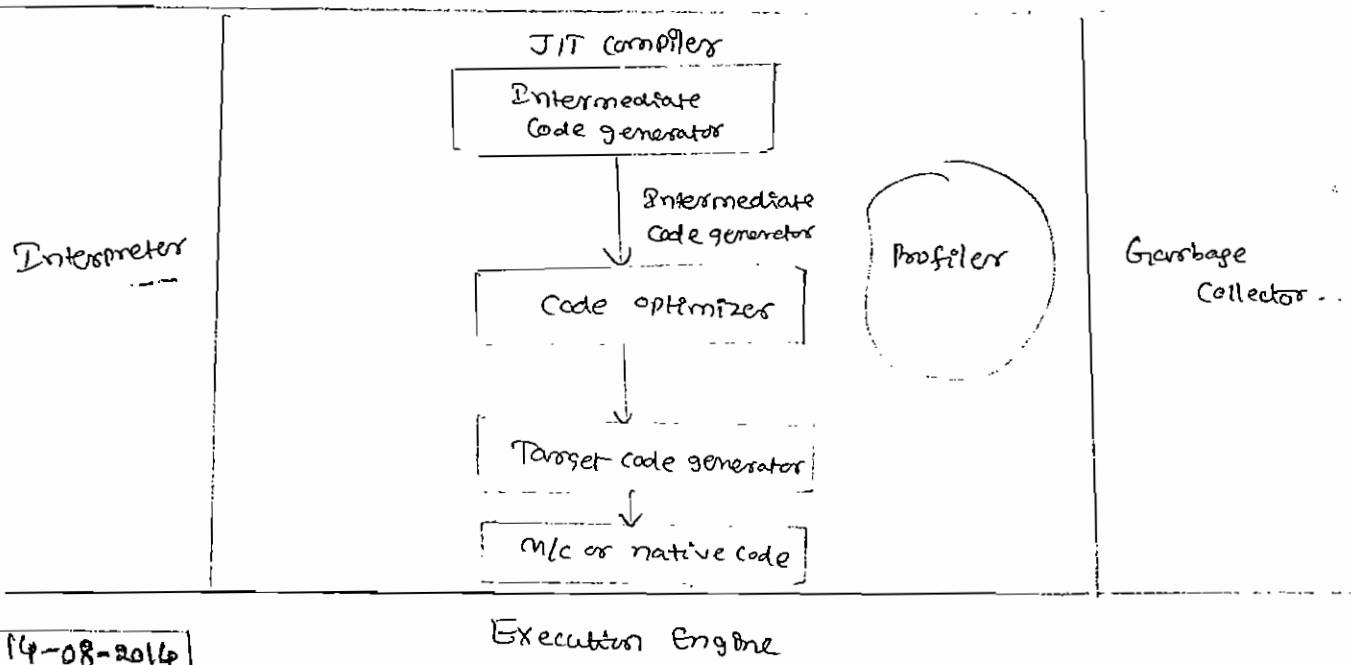
### ② JIT Compiler

- The main purpose of JIT compiler is to improve performance
- Internally JIT compiler maintains a separate count for every method.
- whenever JVM comes across any method call, first that method will be interpreted normally by the interpreter and JIT compiler increments the corresponding count variable. This process will be continued for every method once if any method count reaches threshold value then JIT compiler identifies that method is a repeatedly used method (hotspot) immediately compiles that method and generates corresponding nativecode. Next time JVM comes across that method call then JVM directly uses nativecode and executes it instead of interpreting once again, so that performance of the system will be improved.
- Threshold count varies from JVM to JVM
- Some advanced JIT compilers will recompile generated nativecode if count reaches threshold value second time so that more optimized code will be generated
- Profiler, which is the part of JIT compiler is responsible to identify hot-spots (repeatedly used methods)



- Note:
- JVM interprets total program line by line atleast once.
  - JIT compilation is applicable only for repeatedly <sup>invoked</sup> required method but not for every method.

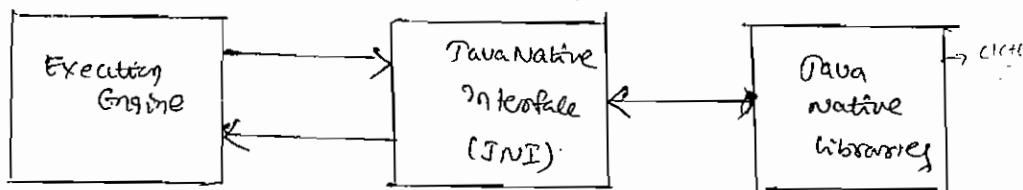
247



### Java Native Interface (JNI)

native libraries

JNI acts as bridge (mediator) for Java method calls and corresponding native libraries.



That is JNI holds and provides information about Native libraries

### Class file Structure (Java)

#### Class file

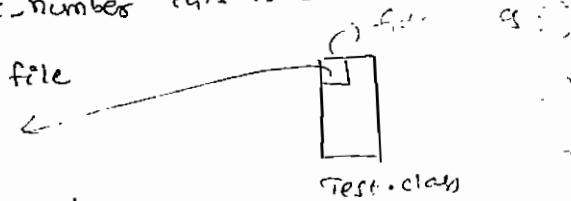
```

magic_number;
minor_version;
major_version;
constant_pool_count;
constant_pool[];
access_flags;
this_class;
super_class;
interface_count;
interface[];
fields_count;
fields[];
method_count;
method[];
attributes_count;
attributes[];
}
  
```

The diagram illustrates the structure of a Java class file. It begins with magic numbers and version information, followed by the constant pool count and its entries. Access flags, this class, and super class are defined. The interface count and its entries follow, along with field and method counts and their respective arrays. The entire structure is enclosed in a brace at the bottom.

magic-number

- The first 4 bytes of class file is magic-number this is a predefined value to identify java class file
  - This value should be **0XCAFEBABE**
  - JVM will use this ~~value~~ magic number to identify whether class file is valid or not
  - That is whether it is generated by valid compiler or not

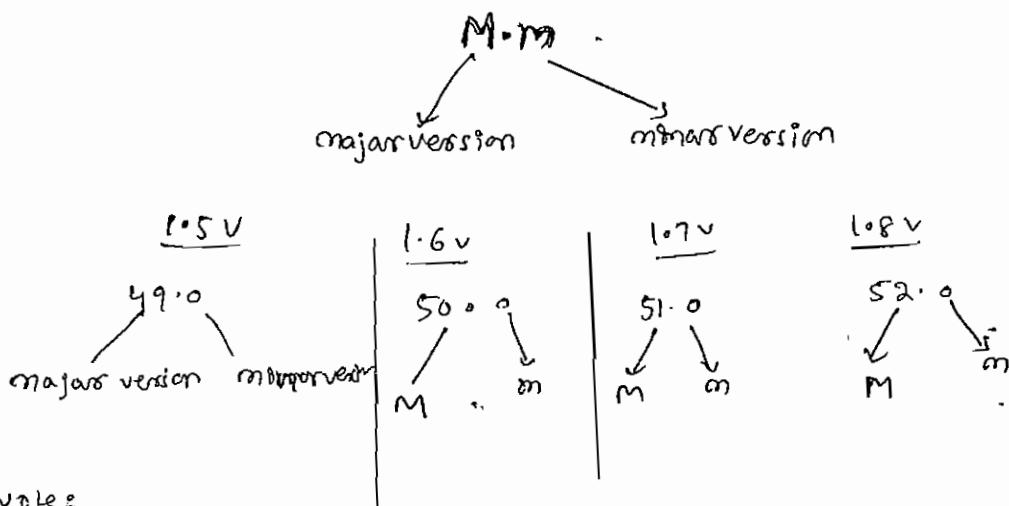


Note 8 whenever we are executing a java class if JVM unable to find magic-number then we will get runtime-exception saying

`java.lang.ClassFormatError : Incompatible magic value`

major & minor version

- Major and minor versions represents classfile version
  - JVM will use these versions to identify which version of compiler generates the current «.class» file.



## Notes

→ Higher version JVM can always run class files generated by lower version compiler but lower version JVM can't run class files generated by higher version compiler.

2) if we are trying to run we will get runtime exception

Saying      UnsupportedClassVersionError: Test  
Unsupported major/minor version

Constant\_Pool\_Count:

$n$  represents number of entries present in constant table of the classfile

constant\_pool []

it represents information about constants present in constant table

access\_flags

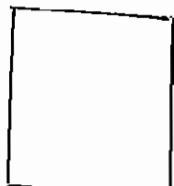
it provides information about modifiers after of the current class

this\_class

it represents fully qualified name of current class

super\_class

it represents fully qualified name of super class



Test.class

this\_class : Test

super\_class : java.lang.Object

interface\_count

it represents number of interfaces implemented by current class

interface []

it represents the names of the interfaces implemented by current class

fields\_count

it represents number of fields present in current class

static\_fields []

it provides fields information present in current class

methods\_count

it represents number of methods present in the current class

methods []

it represents the names of methods present in current class

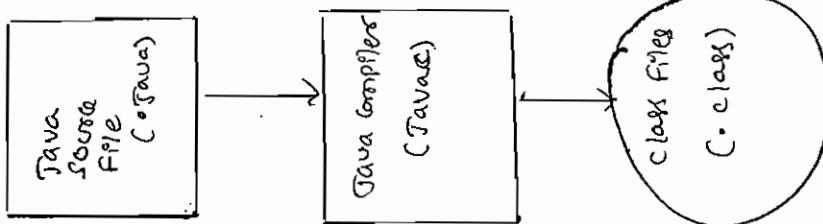
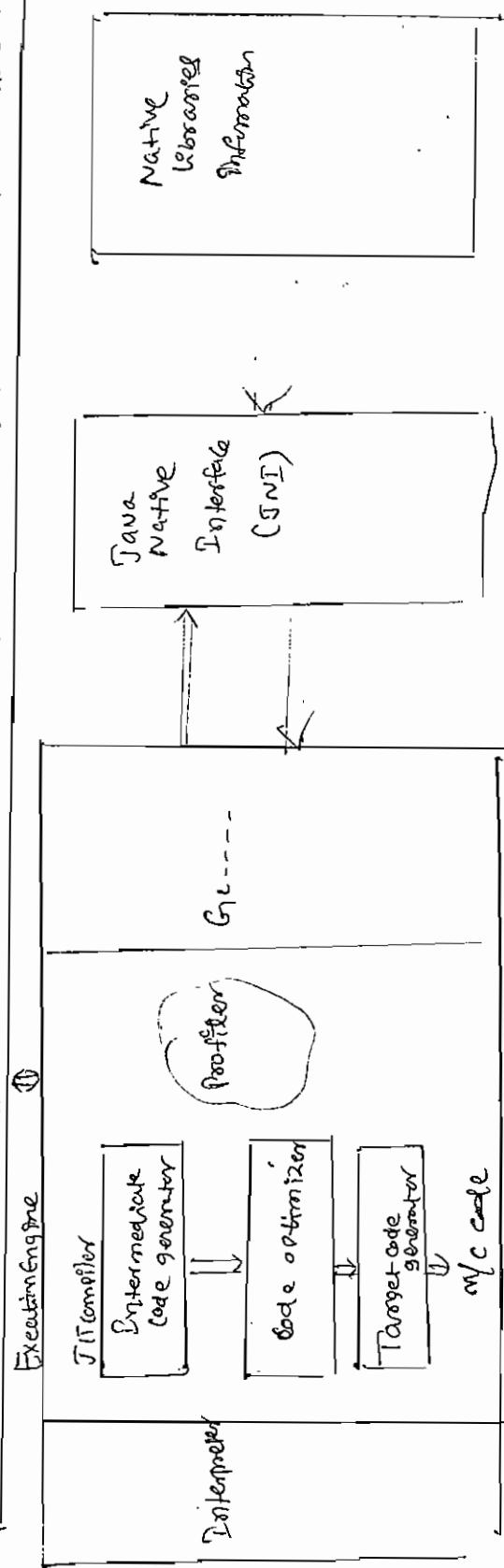
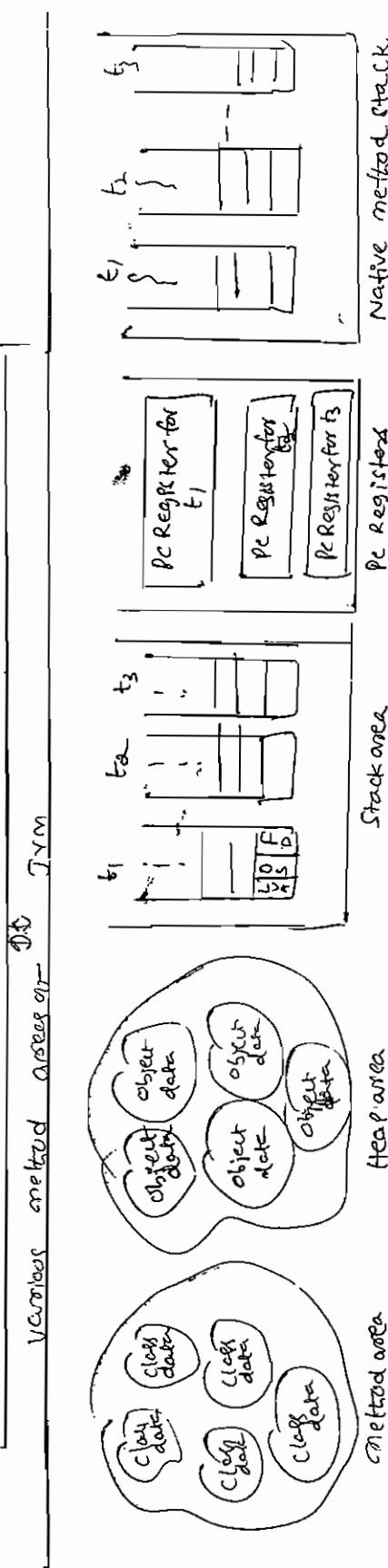
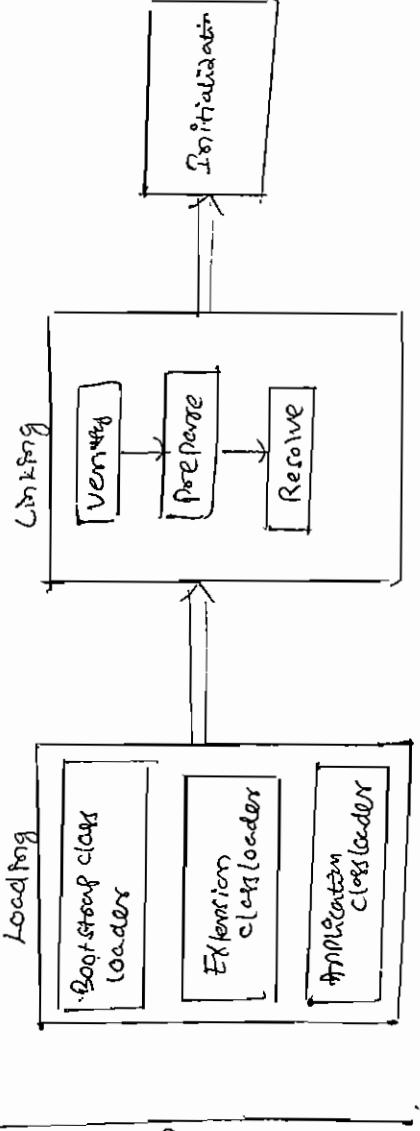
attributes\_count

it represents the number of attributes present in current class

attribute []

it provides information about all attributes present in current class

## Class Loader subsystem



LVA - Local Variable Array  
OS - Operand Stack  
Po - Frame data